FINAL REPORT

Transient Faults in Computer Systems

NASA Grant No. NSG-1442

Gerald M. Masson
Principal Investigator

FINAL REPORT

Primary

Transient Faults in Computer Systems

NASA Grant No. NSG-1442

Gerald M. Masson
Principal Investigator
Department of Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218-2694
Phone: (410) 516-7013
FAX: (410) 516-6134
Email: masson@cs.jhu.edu

Summary

We have developed by means of support from NASA Grant NSG-1442 a novel and powerful technique particularly appropriate for the detection of errors caused by transient faults in computer systems. The technique can be implemented in either software or hardware; the research conducted thus far primarily has considered software implementations. The error detection technique we have developed has the distinct advantage of having provably complete coverage of all errors caused by transient faults that affect the output produced by the execution of a program. In other words, the technique does not have to be tuned to a particular error model to enhance error coverage. Also, the correctness of the technique can be formally verified.

When implemented in software, this new technique uses time and software redundancy and can be outlined as follows. In the initial phase, a program is run to solve a problem and store the result. In addition, this program leaves behind a trail of data which we call a *certification trail*. In the second phase, another program is run which solves the original problem again. This program, however, has access to the certification trail left by the first program. Because of the availability of the certification trail, the second phase can be performed by a less complex program and can execute more quickly. In the final phase, the two results are compared and if they agree the results are accepted as correct; otherwise an error is indicated. An essential aspect of this approach is that the second program must always generate either an error indication or a correct output even when the certification trail it receives from the first program is incorrect. We have formalized the certification trail approach to fault tolerance and have illustrated numerous realizations of it for well-know and important problems. We have rigorously proven the correctness of the technique for certain applications. We have shown cases in which the second phase can be run concurrently with the first and act as a real-time monitor. We have compared the certification trail approach to other approaches to error detection to demonstrate the significant conceptual and performance advantages.

This research has developed the foundation for an effective, low-overhead, software-based certification trail approach to real-time error detection resulting from transient fault phenomena. It would be particularly appropriate at this time to examine the technique further in the context of important and timely applications. For example, transient error phenomena caused by ionizing radiations in space or high-altitude avionics environments stand as a major obstacle to many

applications of high performance microelectronics. The research reported in the following would provide a framework for the development of "radiation-hardened software" that would permit the utilization of high performance microelectronics in space and high-altitude avionics applications in an efficient and cost effective manner.

In the following, seven papers are provided which together characterize the current state of the most recent research conducted with support from NASA Grant NSG-1442:

1. *Certification of Computational Results*, Gregory F. Sullivan, Dwight S. Wilson, Gerald M. Masson.
2. *Experimental Evaluation of the Certification-Trail Method*, Gregory F. Sullivan, Dwight S. Wilson, Gerald M. Masson, Mamoru Itoh, Warren W. Smith, Jonathan S. Kay.
3. *Certification Trails and Software Design for Testability*, Gregory F. Sullivan, Dwight S. Wilson, Gerald M. Masson.
4. *Experimental Evaluation of Certification Trails using Abstract Data Type Validation*, Dwight S. Wilson, Gregory F. Sullivan, Gerald M. Masson.
5. United States Patent, *Method and Apparatus for Fault Tolerance*, Patent No. 5,243,607, Sept. 7, 1993, United States Patent Office.
6. *Using Certification Trails to Achieve Software Fault Tolerance*, Gregory F. Sullivan, Gerald M. Masson.
7. *Certification Trails for Data Structures*, Gregory F. Sullivan, Gerald M. Masson.

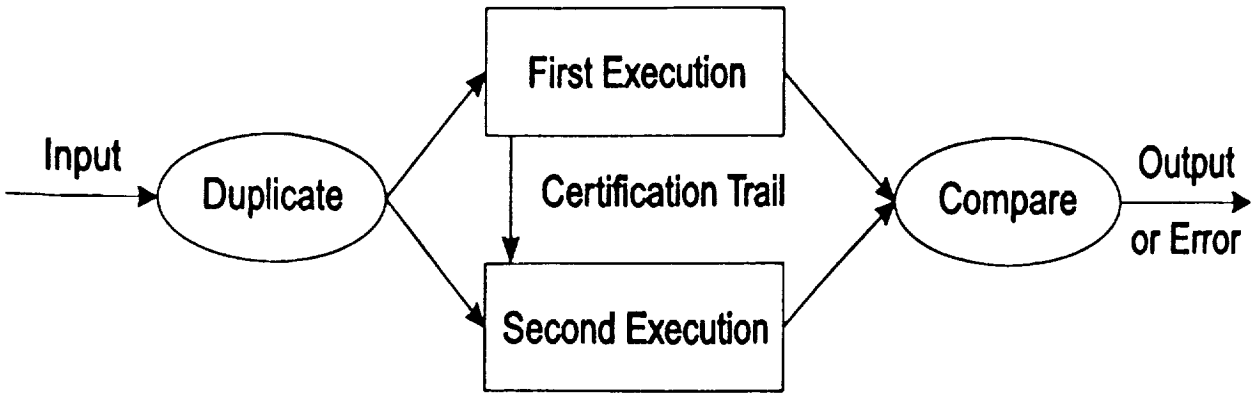


Figure 1: Certification trail method.

the software in addition to those caused by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

A significant drawback to the above approaches is the overhead required. Either extra time is required to run the algorithms serially on a single processor or extra hardware is required to run them in parallel. The technique we will describe is designed to achieve similar types of error detection capabilities while reducing the required resource overhead. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. This data is chosen to allow the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, that we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output produced by the first algorithm. Intuitively, we can regard the two executions as "adversaries." The second execution must guard against an incorrect certification trail "fooling" it into producing an incorrect output. The definitions we give below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the certification trail.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 2.1 A problem **P** is formalized as a relation, i.e., a set of ordered pairs. Let **D** be the domain (that is, the set of inputs) of the relation **P** and let **S** be the range (that is, the set of solutions) for the problem. We say an algorithm **A** solves a problem **P** iff for all $d \in \mathbf{D}$ when d is input to **A** then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 2.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. $\mathbf{T}$ is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that
$F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$
either ($F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$) or $F_2(d, t) = \text{error}$.

We also require that F_1 and F_2 be implemented so that they map elements not in their respective domains to the error symbol. The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, that the examples to be considered will indicate that this approach can also save overall execution time.

The certification trail approach also allows for the detection of faults in software. As in 2-version programming, separate teams can write the algorithms for the first and second executions. Note that the specification now must include precise information describing the generation and use of the certification trail. Because of the additional data available to the second execution, the specifications of the two phases can be very different; similarly, the two algorithms used to implement the phases can be very different. (This will be illustrated in the convex hull example to be considered later.) Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. (This will be illustrated in the shortest path example to be considered later.) When significantly different algorithms are used, the probability that both algorithms will contain or be affected by faults which generate matching errors should be reduced. When very similar algorithms are used it is sometimes possible to save programming effort by sharing program code. For example, the code implementing any data structures needed by the program might be different, while the code that uses the data structure operations would be the same. This approach is well suited for the creation of libraries of fault-tolerant data structures. While this reduces the ability to detect errors in the software it does not change the ability to detect transient hardware errors as discussed earlier. Furthermore, in situations like the above example, it is possible (perhaps even probable) that the majority of software errors will be in the data structure implementation. Thus the ability to detect software errors may not be reduced as much as first imagined.

Throughout this section we have assumed that our method is implemented with software, however, it is clearly possible to implement the method with assistance from dedicated hardware. It is also possible to generalize the basic idea we have suggested. We discuss some of these generalizations in a later section. Finally, we note that a wide variety of approaches to software fault tolerance have been proposed and we contrast our method to the most closely related ideas in a later section.

In the following two sections we illustrate the application of certification trails to three well-known and significant problems in computer science: the convex hull problem, sorting, and the shortest path problem. It should be stressed that the certification trail is not limited to these problems. Rather, these algorithms have been selected for illustrative purposes.

3 Certification Trails for Convex Hulls

The convex hull problem is a fundamental one in computational geometry. Our certification trail solution is based on a solution due to Graham [13] called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos [20]. This text also illustrates some statistical applications of convex hull computations. For simplicity in the following discussion we will assume the points are in so called general position, i.e., no three points are co-linear. It is not difficult to remove this restriction.

Definition 3.1 The *convex hull* of a set of N points, S, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique and its vertices are a subset of the points in S. It is specified by a counterclockwise sequence of its vertices.

The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects the point with minimum x-coordinate (using minimum y-coordinate to break ties), and calls it p_1. For each other point q in S we compute the slope of the line p_1q. Sort the points of S (except for p_1) by this slope (since the points are in general position, the slopes are distinct). Number these vertices $p_2, p_3, \ldots, p_N$. It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although this polygon might not be convex. It is possible to think of the algorithm as removing points from this simple polygon until it becomes convex. This code below performs this by "walking" through the vertices in order. The main FOR loop iteration adds points to the polygon under construction. After a point is added, the inner WHILE loop checks the angle formed by the addition of this point. (Note: We measure angles as follows: Given the three points q_{m-1}, q_m, p_k we measure the angle from $q_{m-1}q_m$ to q_mp_k in the clockwise direction.) If the angle is not acute (i.e., it makes the the polygon non-convex), then the angle vertex (i.e., the preceding point on the polygon) is removed. Note that this will change the preceding angle, which may now be obtuse and should be eliminated. The WHILE loop terminates when an acute angle is encountered. Figure 2 illustrates the construction of a convex hull using this algorithm. from the hull.

When the main FOR loop is complete the convex hull has been constructed.

Algorithm CONVEXHULL(S)
Input: Set of points, S, in R^2
Output: Counterclockwise sequence of points in R^2 which define convex hull of S

```
1  Let p_1 be the point with the smallest x coordinate (and smallest y to break ties)
2  For each point p (except p_1) calculate the slope of the line through p_1 and p
3  Sort the points (except p_1) from the smallest slope to the largest.
   Call them p_2, ..., p_n
4  q_1 := p_1; q_2 := p_2; q_3 := p_3; m = 3
5  FOR k = 4 to n DO
6     WHILE the angle formed by q_{m-1}, q_m, p_k is ≥ 180 degrees DO
7        m := m - 1
8     END WHILE
9     m := m + 1
10    q_m := p_k
11 END FOR
12 FOR i = 1 to m DO, OUTPUT(q_i) END FOR
```

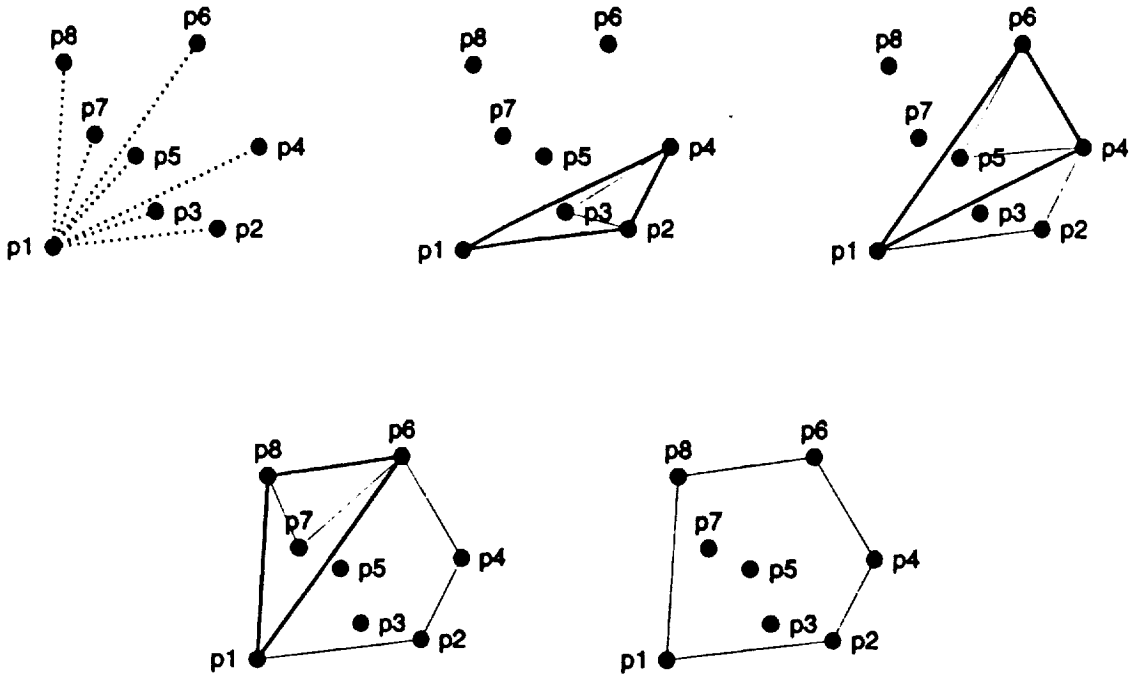


Figure 2: Convex hull example.

END CONVEXHULL

First execution: To generate a certification trail for this algorithm, we rely on the property that for each point eliminated by the WHILE loop in the code above, we can produce a triangle of points in S containing the eliminated point.

Theorem 3.2 *Let p, a, b, and c, be points in the plane such that no three are co-linear, p has the smallest x-coordinate of the four points (and the smaller y-coordinate if another other point has the same x-coordinate) $slope(\overline{pa}) < slope(\overline{pb}) < slope(\overline{pc})$. If the angle abc is obtuse (measured in the clockwise direction), then b is inside the triangle pac.*

Proof: By the ordering of the slopes, b is inside the triangular wedge determined by the rays $\vec{pa}$ and $\vec{pc}$. Note that the line segments pa and pc are in the half plain $x \geq p_x$, and in at least one case the inequality is strict, since no three points are co-linear. This implies that the angle apc (in the clockwise direction) must be greater than 180 degrees. Since the angle abc is also obtuse, both p and b must be on the same side of line $\overline{ac}$. Therefore, b is inside the triangle pac. ∎

Corollary 3.3 *During execution of CONVEXHULL, if, after adding p_k, the angle formed by q_{m-1}, q_m, p_k is obtuse (measured in the clockwise direction), then q_m is contained in the triangle p_1, q_{m-1}, p_k.*

Proof: $slope(\overline{p_1 q_{m-1}}) < slope(\overline{p_1 q_m}) < slope(\overline{p_1 p_k})$. ∎

In the first execution the code CONVEXHULL is used. The certification trail is generated by adding an output statement within the WHILE loop. Specifically, if an angle greater than 180 degrees is found in the WHILE loop test then the 4-tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The table below shows the 4-tuples of points that would be output by the algorithm when run on the example in Figure 2. The points in the table are given the same names as in Figure 2. The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Finally, the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $s_1, s_2, \ldots, s_n$ then rather than output the element in R^2 corresponding to s_i the number i is output. If point coordinates were output instead of these indices, the second execution would have to verify that the points on the trail are members of S.

Point not on convex hull	Three surrounding points
p_3	p_4, p_1, p_2
p_5	p_6, p_1, p_4
p_7	p_8, p_1, p_6

Second execution: Let the certification trail consist of a set of 4-tuples, $(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r)$ followed by the supposed convex hull, $q_1, q_2, \ldots, q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

i. That there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.

ii. That for $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.

iii. For $i \in \{1, \ldots, r\}$ that x_i lies within the triangle defined by a_i,b_i, and c_i.

iv. That for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is acute.

v. That there is a unique point among the points on the supposed convex hull which is a locally maximal point. We say a point q on the hull is a *local maximum* point if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; in fact it makes it easier to verify the first and second conditions.

Time complexity: In the first execution the sorting of the input points takes $O(n \log(n))$ time where n is the number of input points. One can show that this cost dominates and the overall complexity is $O(n \log(n))$.

It is possible to implement the second execution so that all five checks are done in $O(n)$ time. Because indices into the input data are used, the first condition can be checked by verifying that each index is used exactly once, and that all indices are between 1 and N. The second condition may checked simply by verifying that each index is between 1 and N. Checking that a point lies

within a triangle is a geometric calculation that can be done in constant time. Checking that the angle formed by three points is acute requires only constant time. The third and fourth checks can be done in $O(n)$ because the certification trail contains indices into the input data as described above. The uniqueness of the "local maximum" requires only a constant time calculation at each point, so it may checked in linear time.

Experimental timing data for this method may be found in Section 6.

3.1 Proof of correctness

We wish to prove that the algorithms above constitute a certification trail solution for the convex hull problem. Although the definition is phrased in terms of functions, not algorithms, we can simply define the functions $F_1(d)$ and $F_2(d,t)$ on particular arguments as the values computed by the associated algorithms.

Using our formal definition of certification trails, let **D** be the set of all finite planar point sets T. Let **S** be the set of convex polygons, with vertices in counterclockwise order (the restriction to counterclockwise ordering makes the convex hull unique). Then the problem we are considering is $HULL : \mathbf{D} \rightarrow \mathbf{S}$ where $HULL(T)$ is the polygon in **S** that forms the convex hull of T.

The description of the algorithms above defines functions F_1 and F_2. We must show that both conditions of Definition 2.2 hold. The following two lemmas, which we state without proof, are required.

Lemma 3.4 *Let P be a polygon on n points $p_1, p_2, \ldots, p_n$. P is a convex polygon iff P is simple and each angle $p_i p_j p_k$ is less than or equal to 180 degrees, where i is in $1, 2, \ldots n$, $j = (i+1) \bmod n$, and $k = (i+2) \bmod n$.*

Lemma 3.5 *If P is a non-simple polygon, then either P has more than one local maxima, or the interior angle at some vertex is greater than 180 degrees.*

Theorem 3.6 *$F_1(d)$ and $F_2(d,t)$, as defined above, constitute a certification trail solution for the problem HULL.*

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: Recall that the first condition is: for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. Intuitively, this means that if both executions perform correctly, then they will both output the convex hull of the input, which is unique. Note that generation of the certification trail does not affect the output of the Graham Scan algorithm. Thus the condition on $F_1(d)$ is satisfied by the correctness of the Graham Scan algorithm, the proof of which is well known [20]. To show that $F_2(d,t) = s$, note that a copy of s is contained on the trail t. Our description of $F_2(d,t)$ states that s is output unless one of the five checks above fails. It is trivial to verify that the first three of these checks must be satisfied. The fourth check cannot fail, since the polygon described by s is convex (because $(d,s) \in \mathbf{P}$). Similarly, if the fifth check fails, then the polygon described by s has two local maxima, and this is not possible for a convex polygon.

Part 2: The second condition is: for all $d \in \mathbf{D}$ all $t \in \mathbf{T}$ either ($F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$) or $F_2(d,t) =$ error. Intuitively, this means that given an input and arbitrary trail, $F_2(d,t)$ produces a solution to the problem or flags an error. Our definition of $F_2(d,t)$ states that the polygon Q stored on the trail is output unless one of the five checks fails. We must therefore demonstrate that if all five checks succeed, then Q is the convex hull of the input points d. Let H be the convex hull of the points d. The first condition guarantees that every point in d is classified as a hull point or an

interior point. The second condition guarantees that the triangles used to identify interior points are formed from input points, and the third check verifies that the interior points are indeed inside their respective triangles. Note that we do not attempt to verify that the triangles on the trail are the ones that would be produced by $F_1(d)$. In general, for a given interior point, there may be several triangles of input points in which it is contained. Together, the first three conditions imply that all points in H are also in Q, since it is impossible for a hull point to be contained in a triangle. Note that these three checks do not exclude the possibility that interior points are present in Q, nor do they guarantee that the ordering of the hull points in Q is correct. The final two checks will accomplish this. If the last two checks are satisfied, Lemma 3.5 states that Q is simple, and therefore it must be convex by Lemma 3.4.

Thus, Q is a convex polygon whose vertex set is a superset of the vertices of H, i.e., H is contained in Q. This implies that no other point from the input set may be a vertex of Q, since any input point that is not a hull point is interior to H and therefore interior to Q. Finally, it is clear that the ordering of the vertices of Q and H must be the same (although there might appear to be two possible orderings, clockwise and counterclockwise, a clockwise ordering will fail the fourth check). Therefore if all five checks succeed, then the output of $F_2(d,t)$ will be the convex hull of d.

This demonstrates that the algorithms described meet the conditions of Definition 2.2, and are therefore a certification trail solution to the convex hull problem. ∎

3.2 Other convex hull algorithms

It is possible to use this technique to provide certification trails for other convex hull algorithms. The key is that for each non-hull point p we must find a triangle of input points (not necessarily hull points), containing p. For some convex hull algorithms, a containing triangle is available directly or can be easily computed when it is determined that a particular point is not on the hull. However, this is not true of all convex hull algorithms. If, however, we allow extra overhead during the first execution we may apply this technique to any planar convex hull algorithm, provided that the output is a polygon and not merely an unordered list of hull vertices.

Let $H = q_1, q_2, q_3 \ldots, q_h$ be the convex hull of a set of n points. We label the points so that q_1 is the point with smallest abscissae (and smallest ordinate in case of a tie). Since H is convex, the remaining points occur in sorted angular order around q_1. Now for each non-hull point p, we may determine which triangle $p_1 p_i p_{i+1}$ it lies in with a binary search. Thus we may determine containing triangles for the non-hull points in $O(n \log h)$ time. Under several distributions the number of hull points is much smaller than the number of input points [20] so this overhead will often be quite small.

4 Sorting

Sorting is one of the most important basic problems in computer science. There is a massive body of literature discussing sorting and a significant fraction of computer time is spent performing sort operations. We will see how the certification trail approach may be applied to this problem. Assume that a particular sorting algorithm takes as input an array of n elements and outputs an array of n elements. The algorithm is supposed to place the data into non-decreasing order.

Note that it may not appear necessary to use a certification trail for this problem. It might seem that all that is required is to verify that the output is in non-decreasing order. Unfortunately, this is not sufficient and we must also verify that the output consists of the same elements as the input. A certification trail is required to perform this check efficiently.

The information placed on the trail is a permutation relating the input and output arrays. This permutation is created by adding an Item Number field to the elements being sorted, such that the i-th element is labelled with item number i. After sorting, the permutation is obtained by reading the Item Numbers from the elements in their new order.

The second algorithm reads the permutation from the trail, uses it to rearrange the input elements in linear time, and checks that they are now in sorted order. Additionally, it is necessary to check that the the information on the certification trail actually is a permutation of n elements, i.e., each number from 1 to n occurs exactly once. Should any of these checks fail, the second algorithm outputs "error", otherwise it outputs the sorted elements.

Note that the certification trail given for sorting is quite different than that given for the convex hull problem. In the latter case, the certification trail was constructed for a particular algorithm, and the code executing that algorithm modified to produce the trail. In this case, the sorting algorithm is not changed. Instead the data being sorted is modified by a preprocessing step, and the necessary information extracted by a postprocessing step. Thus this technique may be implemented as a "wrapper" around existing sort routines, no matter which algorithm is implemented.

Experimental data is presented in Section 6.

4.1 Proof of correctness

For concreteness we consider only the sorting of integers, though the proof does not depend on this condition.

Definition 4.1 Let $\mathbf{D}$ consist of all finite sequences of integers. Let $\mathbf{S}$ consist of all finite non-decreasing sequences of integers. Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be the sorting problem, i.e., $(d, s) \in \mathbf{P}$ iff s is a permutation of d (by definition of $\mathbf{S}$, s is a non-decreasing sequence). Note that for every $d \in \mathbf{D}$, there is a unique $s \in \mathbf{S}$ such that $(d, s) \in \mathbf{P}$. Let $\mathbf{T}$ consist of finite sequences of integers. For x a member of any of the sets $\mathbf{D}, \mathbf{S}$, or $\mathbf{T}$, we will also denote the sequence of integers by $x_1, x_2, ..., x_N$.

Definition 4.2 The function $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ is defined as follows. Given an input sequence d of N integers, $F_1(d) = (s, t)$ where s is the unique element of $\mathbf{S}$ such that, $(d, s) \in \mathbf{P}$ and t is a permutation of 1,2,3,...,N s.t., $s_i = d_{t_i}$ for all $i = 1, 2, ...N$. Note that unless d consists of N distinct integers, there will be more than one possible t. The t produced by $F_1(d)$ may be chosen arbitrarily. Since for every $d \in \mathbf{D}$, there exists a unique $s \in \mathbf{S}$ with $(d, s) \in \mathbf{P}$, the function F_1 is well defined.

Definition 4.3 The function $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$ is defined as follows. $F_2(d, t) = d_{t_1}, d_{t_2}, ..., d_{t_N}$ (where d consists of N integers) iff

i. t contains at least N integers.

ii. The first N integers of t are a permutation of $\{1, 2, ...N\}$.

iii. $d_{t_i} \leq d_{t_{i+1}}$ for $i = 1, 2, ..., N - 1$.

Otherwise, $F_2(d, t) =$ error. Note that though t may contain more than N integers, $F_2(d, t)$ depends only on the first N.

The definitions of the functions F_1 and and F_2 correspond to the informal descriptions of the sorting algorithms given in the text above.

Theorem 4.4 *F_1 and F_2 are a certification trail solution to the sorting problem* $\mathbf{P}$.

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: We must prove that for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. If $F_1(d) = (s,t)$, then by definition $(d,s) \in \mathbf{P}$. We must show that $F_2(d,t) = s$. t is a permutation of $\{1,2,...,N\}$, so the first two conditions of Definition 4.3 are satisfied. Furthermore, by Definition 4.2, $d_{t_i} = s_i$ for $i = 1,2,...N$. Since $s \in \mathbf{S}$, it is a nondecreasing sequence, and thus the third condition of Definition 4.3 is satisfied. Therefore $F_2(d,t) = s$.

Part 2: We must show that for all $d \in \mathbf{D}$ and all $t \in \mathbf{T}$ either ($F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$) or $F_2(d,t) =$ error. Pick $d \in \mathbf{D}$ with length N. Pick $t \in \mathbf{T}$. The interesting case is when t is a permutation of $\{1,2,...,N\}$. If not, then either the first N integers of t are not such a permutation, in which case $F_2(d,t) =$ error. We may ignore the possibility that t consists of such a permutation followed by more integers, since F_2 depends only on the first N integers of t.

Examine the sequence $d_{t_1}, d_{t_2},,, d_{t_N}$. If there is an i such that $d_{t_i} > d_{t_{i+1}}$ then the third condition of Definition 4.3 is violated so $F_2(d,t) =$ error. Otherwise $F_2(d,t) = d_{t_1}, d_{t_2}, ..., d_{t_N}$. Furthermore, this is a non-decreasing sequence, so it must be in $\mathbf{S}$. Finally, since this sequence is a permutation of d, $(d, F_2(d,t)) \in \mathbf{P}$.

Therefore, both conditions of Definition 2.2 are satisfied, so F_1 and F_2 constitute a certification trail solution to sorting. ∎

Note that we defined $\mathbf{T}$ as the set of all finite sequences of integers. We could have instead defined $\mathbf{T}$ as the set of permutations of $\{1,2,...N\}$ for all positive N. This would make the function F_2 "simpler", in that it doesn't have to verify that that certification trail consists of a permutation (it would, however, have to verify that it consists of a permutation of the correct size). In this case, checking that the trail t is indeed a permuation (i.e., actually in its domain) would be left to the implementation of the function.

5 Certification Trails for Shortest Paths

This classic problem has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [11] as explicated in [10]. First we require some preliminary definitions.

Definition 5.1 A *graph* $G = (V,E)$ consists of a *vertex* set V and an *edge* set E. An edge is an unordered pair of distinct vertices which we notate with the following style: $[v,w]$ and we say v is adjacent to w. A *path* in a graph from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $[v_i, v_{i+1}]$ is an edge for $i \in \{1, \ldots, k-1\}$. Let w be a real function defined on E. The *length* of a path from v_1 to v_k is the sum of $w([v_i, v_{i+1}])$ for each edge $[v_i, v_{i+1}]$ in the path.

Let $G = (V, E)$ be a graph and let w be a positive rational valued function defined on E. Given a vertex v_1 in V, find a set of shortest paths from v_1 to each other vertex in V. Note that since w is positive on all edges, a shortest path must exist between any two vertices, though it need not be unique.

Before we discuss the algorithm we must describe the properties of the principal data structure that are required. Since many different data structures can be used to implement the algorithm, we initially describe abstractly the data that can be stored by the data structure and the operations that can be used to manipulate this data. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the *item number* and the second element is called the *item value* or just *value*. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for

multiple ordered pairs to have the same item value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, x is a value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i,x) < (i',x')$ iff $x < x'$ or $(x = x'$ and $i < i')$. Our data structure should support a subset of the following operations.

member(i,h) returns a boolean value of true if h contains an ordered pair with item number i, otherwise returns false.

insert(i,x,h) adds the ordered pair (i,x) to the set h.

delete(i,h) deletes the unique ordered pair with item number i from h.

changekey(i,x,h) is executed only when there is an ordered pair with item number i in h. This pair is replaced by (i,x).

deletemin(h) returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If h is the empty set then the token "empty" is returned.

predecessor(i,h) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned.

A description such as the one above describes an *abstract data type*. There may be several possible implementations for a particular ADT. In our solution, different ADT implementations will be used for the two executions. The first implementation will produce a certification trail allowing the second implementation to be simpler and to perform ADT operations more quickly.

Aside from the implementation of the abstract data type, both of our algorithms are the same. Pidgin code for this algorithm appears below. Figure 3 illustrates the execution of the algorithm on a sample graph. Table 1 records the data structure operations performed when the algorithm is run on the sample graph. The first column gives the operations, with the parameter h omitted to reduce clutter. Member operations are also omitted from the table. The second column gives contents of h after the execution of each instruction. The third column records the order pair deleted by deletemin operations. The fourth column records the information (if any) output to the certification trail by this operation.

This certification trail is created by modifying the insert(i,x,h) and changekey(i,x,h) operations performed during the first execution. The modified instructions perform the same operations described above and in addition output the following information to the certification trail.

insert(i,x,h) Output the item number of the predecessor of (i,x) (as defined above) to the trail. If there is no predecessor, output the token "smallest". Note that depending on the data structure implementation, the predecessor may already be computed during insertion or may require a separate call to the predecessor(i,h) operation.

changekey(i,x,h) Output the predecessor of the ordered pair (i,x) (i.e., pair resulting from the change) to the trail. If there is no predecessor, output the token "smallest" to the trail.

We shall see that this information allows a faster and simpler data structure implementation to be used for our second algorithm.

The algorithm proceeds by maintaining a set S of vertices for which shortest path lengths are known, and a "frontier" set F of vertices adjacent to members of S along with the best known path

length from v_1. At each step, we find the vertex v in F with smallest known path length and place it in S, F is then updated by examining the neighbors of v. New vertices may be added to F or a shorter path (passing through v) may be found to existing vertices in F.

To efficiently find the vertex to add to S, the algorithm uses the data structure operations described above. As soon as a vertex v is adjacent to some vertex u in S, it is inserted in the set F. The value for v is the shortest known path to v, which is the value of u (shortest path to u) plus the weight of edge vw. The array element prefer(v) is used to keep track of this "best" edge connecting v to S. As the tree grows, information is updated by operations such as insert(i, x, h) and changekey(i, x, h). The deletemin(h) operation is used to select the next vertex to add to the span of the current tree. Note, the algorithm does not explicitly store paths. Implicitly, however, if (v, x) is returned by deletemin, then prefer(v) indicates the predecessor of v on the shortest path from v_1.

Algorithm SHORTEST-PATH(G,v_1,weight)
Input: Connected graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ with edge weights.
Output: Lengths of shortest paths from v_1 to all other vertices.

```
1  FOR ALL u ∈ V, u) := ∞ END FOR
2  v1) := 0
3  F := v1;
4  WHILE F ≠ ∅ DO
5   (v, k) := deletemin(F)
6   FOR EACH [v, w] ∈ E DO
7    IF v) + weight([v, w]) < w) THEN
8     w) := v) + weight([v, w]); prefer(w) := v
9     IF member(w, F) THEN changekey(w, w), F)
10     ELSE insert(w, w), F) END IF
11   END IF
12  END FOR
13 END WHILE
14 FOR ALL u ∈ V − {v1}, OUTPUT(u)) END FOR
END SHORTEST-PATH
```

Note that this code may be easily modified to output the shortest paths as well as their lengths.

First execution: In this execution the SHORTEST-PATH code is used and the abstract data type is implemented with a balanced search tree such as an AVL tree [1], a red-black tree [14], or a b-tree [5]. In addition, an array indexed from 1 to n is used. Each element of this array contains two fields, *InSet*, a boolean, and *Value*, storing the same type as the value used in the ordered pairs. Initially, *InSet* is false for all array elements. The balanced search tree stores the ordered pairs in h and is based on the total order described earlier. For each item number i, the *InSet* field of the i-th array element is true if and only if there is a pair with item number i in the set. The *Value* field of the i-th array element stores the value of the pair with item number i, if there is one in the set. It is undefined if there is no such pair in the set. This array allows rapid execution of operations such as member(i, h) and delete(i, h).

Second execution: This execution also uses the SHORTEST-PATH code, however, a different data structure is used to implement the ADT. We call this data structure an *indexed linked list* and it is depicted in Figure 5. It consists of an array and a doubly linked list. The array is indexed from 0 to n and contains pointers to the elements of the linked list. Except for the first element,

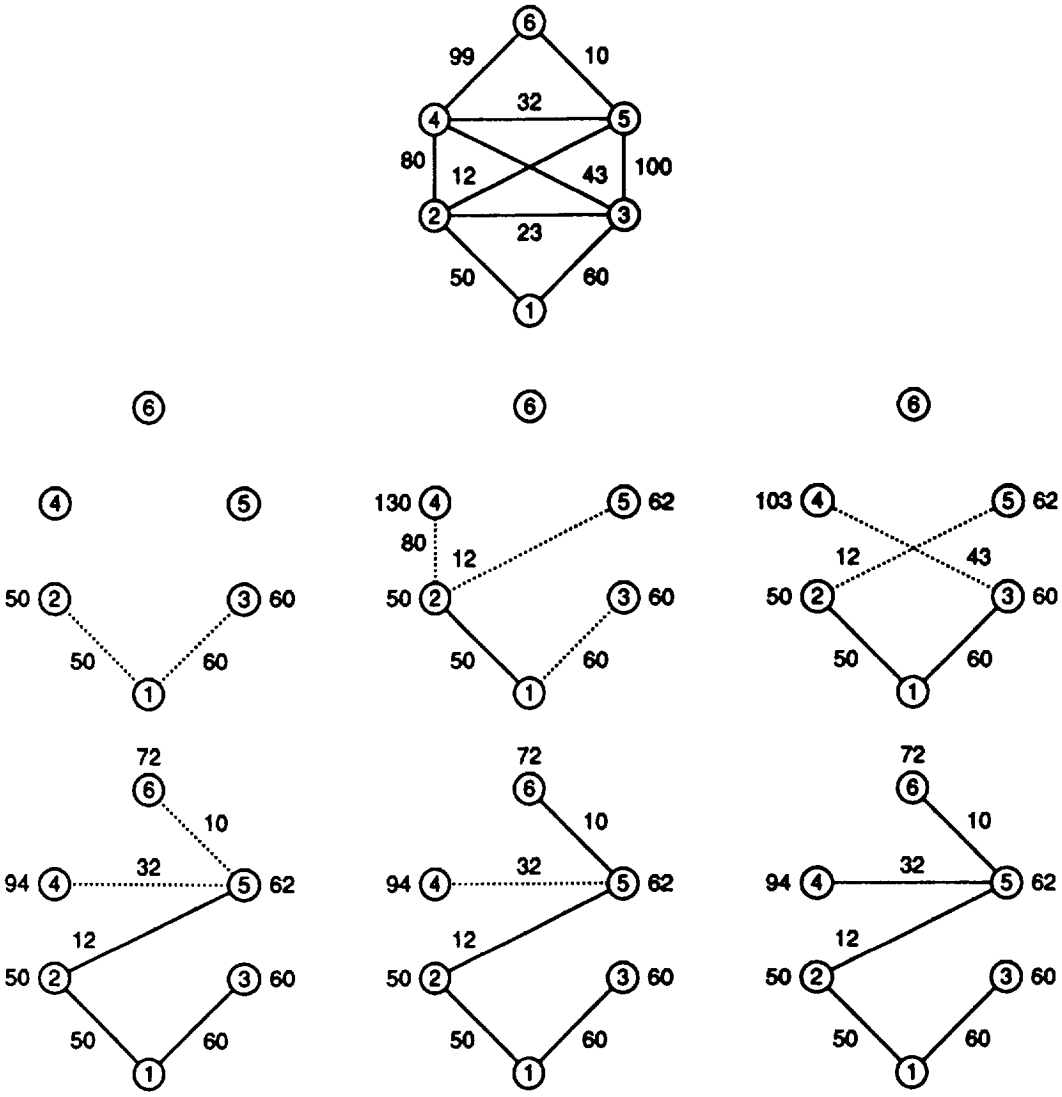


Figure 3: Shortest path example.

Operation	Set of Ordered Pairs	Delete	Trail
insert(2,50)	(2,50)		smallest
insert(3,60)	(2,50),(3,60)		2
deletemin	(3,60)	(2,50)	
insert(4,130)	(3,60),(4,130)		3
insert(5,62)	(3,60),(5,62),(4,130)		3
deletemin	(5,62),(4,130)	(3,60)	
changekey(4,103)	(5,62),(4,103)		5
deletemin	(4,130)	(5,62)	
changekey(4,94)	(4,94)		smallest
insert(6,72)	(6,72),(4,94)		smallest
deletemin	(4,94)	(6,72)	
deletemin		(4,94)	
deletemin		empty	

Table 1: Example of operations and trail.

each element in the list contains a data field storing an ordered pair. The first element stores a special ordered pair $(0, \text{"}smallest\text{"})$ which is guaranteed to compare less than any other ordered pair. The list is maintained in sorted order based on the total ordering defined above for ordered pairs. This list represents the contents of the set h. The i-th element of the array points to the node containing the ordered pair with item number i, if such an element is present in h. Otherwise the pointer is nil. The 0-th element of the array points to the node containing $(0, \text{"}smallest\text{"})$ Initially, all pointers are nil except for the 0-th one. Using an ordered list allows us to perform deletemin(h) operations quickly. The array provides rapid random access to the elements. We now describe the implementation of the data structure operations.

insert(i, x, h) Read the next value from the certification trail. This value, call it j, is the item number of the ordered pair that will be the predecessor of (i, x) after it is inserted. To insert this element, we follow the j-th array pointer to the list node containing the pair (j, y). There is one special case, if "smallest" is read from the trail rather than an item number, we follow the 0-th pointer. A new node is allocated and inserted into the list just after the node containing (j, y). The data field of this node is set to (i, x). Finally, the i-th pointer is set to point to the new node. Figure 5 shows the insertion of (5,62) into the data structure, given that the next item on the certification trail is 3. When the insert(i, x, h) operation is performed, some checks must be conducted:

i. The i-th array element must be nil before the operation is performed.

ii. The value j read from the trail must either be "smallest" or be between 1 and n, i.e., it must be a valid item number.

iii. The j-th array element must not be nil before the operation is performed.

iv. The sorted order of the pairs stored in the linked list must be maintained. That is, if the j-th pointer points to (j, y) and its successor before the insertion (ignoring the

special case when (j, y) is the last element of the list) is (j', y'), then we must have $(j, y) < (i, x) < (j', y')$.

If any of these checks fails, then the execution halts and "error" is output.

delete(i, h) If the i-th pointer is nil, halt execution and output "error". Otherwise follow the i-th pointer to find the list node containing (i, x). This node is removed from the list. Note that since the list is doubly linked, this is a constant time operation. The i-th pointer is then set to nil. The only condition that must be checked is that the i-th pointer is not nil before the deletion

changekey(i, x, h) To perform this operation, it suffices to perform delete(i, h) followed by insert(i, x, h). The next item for the certification is read when the insert(i, x, h) operation is performed. If any of the conditions required by either of these operations fails, then execution halts and "error" is output.

deletemin(h) The 0-th array pointer is traversed to the list head (which contains $(0, \text{"}smallest\text{"})$). The pointer to the next node in the list is followed. If there is no next node then "empty" is returned. Otherwise, let (i, x) be the pair stored in that node. We remove the node from the list, set the i-th array element to nil, and return (i, x).

member(i, h) The i-th array pointer is examined. "False" is returned if it is nil, otherwise "true" is returned.

predecessor(i, h) This operation is not used during the second execution of SHORTEST-PATH, but is described for completeness. Follow the i-th pointer to the node containing the pair (i, x). Follow the pointer from that node to the node preceding it on the list (note that this node will always exist). If this is the special node $(0, \text{"}smallest\text{"})$, return "smallest", otherwise return the item number of the pair stored in this list.

There are two variations to this scheme that are worth noting. First, we could implement a singly linked list rather than a doubly linked list. This eliminates the overhead of maintaining the extra pointer. Note, however, that operations such as delete(i, h) require access to predecessors in order to update the list quickly. This can be provided by modifying the operations delete(i, h), changekey(i, x, h), and predecessor(i, h) so that they output predecessor information to the trail.

The other variation also uses a singly linked list but removes the need for extra certification trail information for delete(i, h) and changekey(i, x, h) operations. It uses the technique of marking a list node for deletion rather than removing them from the list node immediately (the appropriate pointer in the array is still set to nil immediately). When performing other operations, we check for and remove any marked nodes immediately following nodes visited. The total running time is still linear, though insert operations are no longer constant time operations.

Time complexity: In the first execution each data structure operation can be performed in $O(\log(n))$ time where $|V| = n$. There are at most $O(m)$ such operations and $O(m)$ additional time overhead where $|E| = m$. Thus, the first execution can be performed in $O(m \log(n))$ In addition, it provides us with a relatively simple and illustrative example of the use of a certification trail.

In the second execution each data structure operation can be performed in $O(1)$. There are still at most $O(m)$ such operations and $O(m)$ additional time overhead. Hence, the second execution can be performed in $O(m)$ time, i.e., linear time.

Section 6 contains results of timing experiments with this technique.

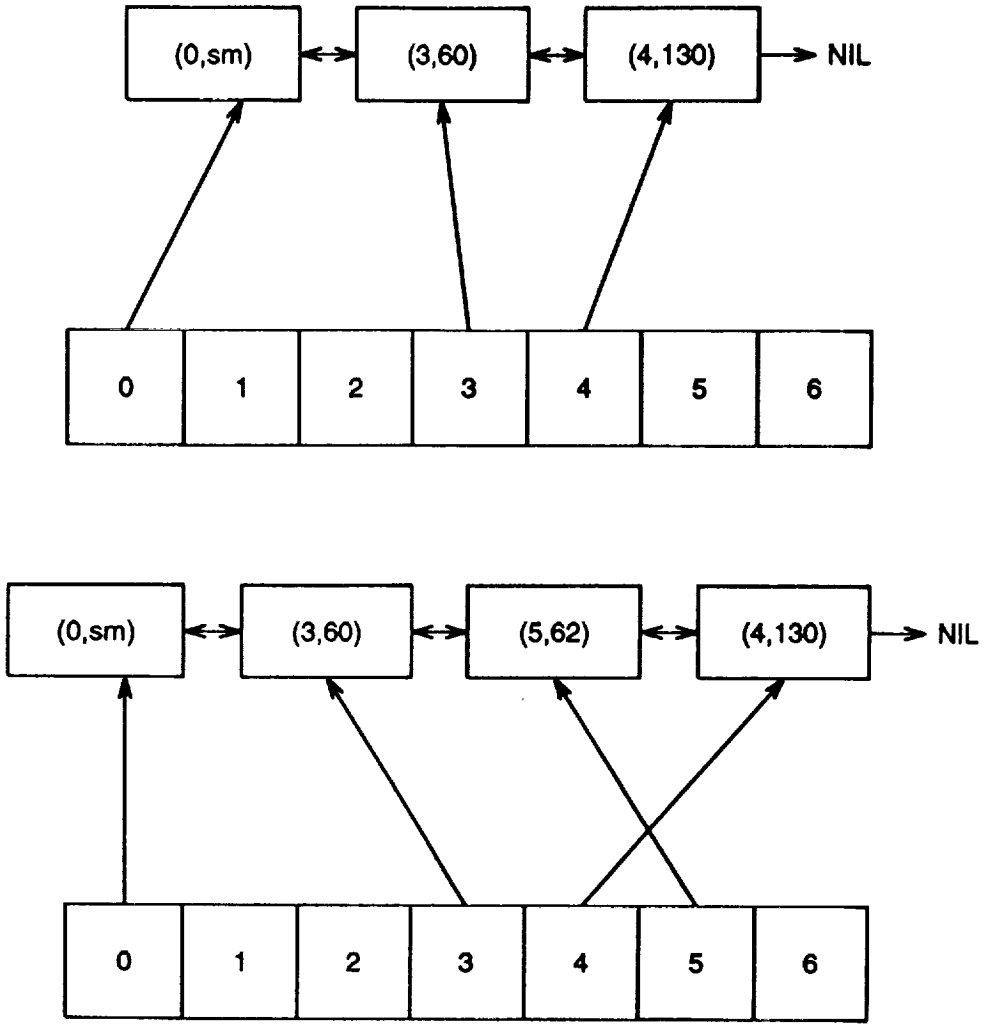


Figure 4: Example of the indexed linked list before and after inserting (5,62)

5.1 Proof of correctness

We wish to prove that the two algorithms given above constitute a certification trail solution to the SHORTEST-PATH problem, i.e., that the functions $F_1(d)$ and $F_2(d,t)$ defined by these algorithms satisfy Definition 2.2. First, we consider the problem of evaluating a sequence of the above data structure operations.

Definition 5.2 Let **D** be the set of finite sequences of the data structure operations defined above. Let **S** be the set of finite sequences of answers to data structure operations. Let **P** be the relation (d,s) where $d \in \mathbf{D}$ and $s \in \mathbf{S}$, and s is the sequence of answers resulting from executing the operations d starting with the empty set.

Note that we are examining all finite sequences of data structure operations, not just "legal" ones. That is, may attempt to perform an insertion with an item number already in use, attempt to perform deletion on an item number not being used, etc. We assume that if one of these "illegal" operations is attempted, the operation will output "error" and terminate processing. Thus, we can define the answer sequences for these "illegal" sequences.

Definition 5.3 Let $F_1(d)$ be defined by the result of executing the operations on any of the standard data structures described above, with the insert(i,x,h) and changekey(i,x,h) operations modified to output trail information. Let $F_2(d,t)$ be defined by the result of executing the operations using the indexed linked list implementation described above.

Theorem 5.4 *$F_1(d)$ and $F_2(d,t)$ meet the conditions of Definition 2.2 (that is, $F_1(d)$ and $F_2(d,t)$ constitute a certification trail solution for* **P***).*

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: The first condition we must verify is that for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. Let $(s,t) = F_1(d)$. The modifications of the data structure operations that produce trail output do not affect how the data structure is maintained. Proofs of correctness for the standard data structures are well known, so we may assume $(d,s) \in \mathbf{P}$. We must demonstrate that $F_2(d,t) = s$.

This may be proven by showing that after each operation that modifies the set h, the elements stored in the indexed linked list (our implementation) correspond to the elements in the set h (the abstract definition). We must also demonstrate that if this relationship is maintained, then correct output is generated by operations that generate output.

To demonstrate this, we show that each operation maintains the following invariants.

i. If the pair (i,x) is in $h \cup (0, \text{"}smallest\text{"})$, then the i-th pointer in the array of pointers points to the list node containing (i,x).

ii. If, for some i, there is no pair in h with item number i then the i-th pointer is nil.

iii. The list nodes are in ascending order.

iv. Every list node is pointed to by some pointer in the array. (Together with the first condition, this implies that it is pointed to by exactly one pointer from the array).

The first two conditions assert that the indexed linked list and the set h contain the same elements (ignoring the special list head element in the linked list). The last two invariants allow us to demonstrate that the linked list operations function correctly.

Clearly each of these conditions is true before the first operation is performed (the set of pairs is empty, all pointers except the 0-th are nil, and $(0, \text{"}smallest\text{"})$ is the only list node).

Assume that the above conditions are satisfied after the first k operations, and that the output generated by any of the first k operations is correct. We claim that the invariants will will remain satisfied after the $(k+1)$-st operation, and that if the $(k+1)$-st operation generates output, it will be correct. Let $s(k+1)$ denote the output produced by the $(k+1)$-st operation (where $F_1(d) = (s, t)$).

Consider each possible operation. For brevity, we omit details for "illegal" operations, i.e., those that violate the precondition of the operation. Similarly, we omit details of the special case of "smallest" being read from the trail.

insert(i, x, h) The trail t contains the item number j of the predecessor of (i, x). Call the predecessor (j, y). By assumption, the i-th pointer is nil before the insert. If not, this operation outputs "error" and execution halts. Since the indexed linked list correctly represents h at this point, this agrees with the result returned by $F_1(d)$, i.e., $s(k+1) = \text{"}error\text{"}$. After the insertion is performed, the i-th pointer is set to the new node containing (i, x), so the first condition is satisfied. No other nodes are added to the list, so the second condition will remain true. The third condition is satisfied since (j, y) is now the immediate predecessor of (i, x). Since no other pointer in the array has been changed, the fourth condition is still true.

delete(i, h) This operation sets the i-th pointer to nil, and removes the node containing (i, x) from the list. This satisfies the second invariant. Deleting a node cannot violate the third invariant. Since no other nodes are removed and no other pointers are changed, the first and fourth invariants remain satisfied.

deletemin(h) By assumption, the nodes are currently in ascending order. Thus, the minimum element in h must correspond to the node following the special list head node, call the pair it contains (i, x). This pair is the correct output for this operation. As with delete, the above four conditions remain true after this node is removed and the i-th pointer set to nil.

changekey(i, x, h) We have implemented changekey(i, x, h) as an insertion followed by a deletion. Since both of those preserve the invariants, changekey(i, x, h) must do so as well.

member(i, h) By assumption, the indexed linked list correctly represents h before this operation, so the output of this operation will be correct. Since this operation does not change the set or the indexed linked list, the invariants remain satisfied.

predecessor(i, h) By assumption, the indexed link list correctly represents h, and furthermore it is currently in sorted order. Thus, the list element preceding the node containing (i, x) is the predecessor. Since this operation changes neither h nor the indexed linked list, the invariants remain satisfied.

This demonstrates that the first condition of Definition 2.2 is satisfied.

Part 2: The second condition is for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$ either $(F_2(d, t) = s$ and $(d, s) \in \mathbf{P})$ or $F_2(d, t) = \text{error}$. Intuitively, this states that if $F_2(d, t)$ is passed an arbitrary trail, it either outputs a correct answer, or it outputs "error". We prove an even stronger condition. Let $t_{correct}$ be the trail returned by $F_1(d)$, i.e., $F_1(d) = (s, t_{correct})$. Then either $t_{correct}$ is a prefix of t, or $F_2(d, t) = \text{error}$.

If $t_{correct}$ is a prefix of t, then we are done. The algorithm describing $F_2(d, t)$ does not examine any part of the trail after $t_{correct}$, so $F_2(d, t) = s$.

If $t_{correct}$ is not a prefix of t, let p be the position at which they first differ. Let O be the number of the operation that uses the trail data at p. Then operation O is either an insert(i, x, h) or changekey(i, x, h) operation. If it is an insert operation, then $t_{correct}$ contains the item number of the predecessor of (i, x). Since t contains a different value, call it j, at this location, the insert(i, x, h) operation will fail one of it's three checks. Either j will not be valid item number, or the j-th pointer will be nil, or the pair (j, y) will not be the predecessor of (i, x). The argument for the changekey(i, x, h) operation is essentially the same.

Thus, the second condition is satisfied.

Therefore, $F_1(d)$ and $F_2(d, t)$ are a certification trail solution to **P**, the problem of evaluating data structure operations. ▮

Definition 5.5 Let **D** be the set of finite graphs $G = (V, E)$ with edge weights consisting of positive integers. Assume the indices are numbered 1 through n. Let **S** be the set of finite ordered tuples of positive integers. Let **P** be the relation that associates each graph with the tuple consisting of the minimum path lengths to each vertex. Let $SP_1(d)$ be the function defined by the SHORTEST-PATH algorithm with the data structure defined for the first execution. Let $SP_2(d, t)$ be the function defined by the SHORTEST-PATH algorithm using the indexed linked list implementation.

Corollary 5.6 *$SP_1(d)$ and $SP_2(d, t)$ constitute a certification trail solution for* **P**.

Proof: If $SP_1(d) = (s, t)$, then the correctness of Dijkstra's algorithm implies that $(d, s) \in$ **P**. The algorithms that compute $SP_1(d)$ and $SP_2(d, t)$ are the same except for data structure implementation. Theorem 5.4 implies that if these algorithms generate the same data structure operations, then the same sequence of answers will be generated. Thus, to demonstrate that $SP_2(d, t) = s$, it must be shown that the same sequence of data structure operations is generated by both algorithms. Examination of SHORTEST-PATH indicates that the k-th data structure operation to be performed is dependent only on the input and the result of previous data structure operations. For example, at line 9, either an insert(i, x, h) or a changekey(i, x, h) is performed, depending on the result of a member(i, h) operation. The input graph d is identical for both algorithms, thus the first data structure operation performed must be the same. Assume that the first k operations performed by both algorithms are identical. Then, by Theorem 5.4, the answers to those operation will be the same. Since the $(k + 1)$-st operation depends only on the input and the results of the previous k operations, it must also be the same for both algorithms. Therefore the same sequence of data operations is performed in both algorithms, so $SP_2(d, t) = s$.

The proof that the second condition holds is the same as for Theorem 5.4. Either the input trail t contains the "correct" trail as a prefix, or one of the data structure operations will fail, resulting in an "error" output. ▮

One point has been glossed over in the above proof. In the SHORTEST-PATH algorithm results of deletemin(h) are not output nor are they stored in the certification trail. It might be possible for incorrect answers to be returned by deletemin(h) operations while still producing correct shortest paths and lengths. The second execution of the SHORTEST-PATH algorithm will not detect this since the correct output is produced. By proving that the answers to deletemin(h) operations are the same, we have proven more than strictly required.

6 Experimental Data on Certification Trails

We have performed extensive timing experiments on several basic and well-known problems, including the ones described in this paper. Algorithms for solving these problems were implemented, both

with and without the use of certification trails. Timing data was collected on both the certification trail solutions and the basic solutions. The following tables summarize these results.

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
5000	0.61	0.62	0.07	8.73	43.62
10000	1.33	1.34	0.14	9.56	44.54
25000	3.68	3.68	0.36	10.22	45.12
50000	7.68	7.74	0.71	10.75	44.94
100000	16.23	16.30	1.43	11.35	45.39
200000	33.93	34.37	2.84	11.94	45.16

Table 2: Convex Hull

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
10000	0.28	0.30	0.04	7.00	39.29
50000	1.80	1.90	0.19	9.47	41.94
100000	3.96	4.08	0.41	9.66	43.31
500000	23.95	24.69	2.14	11.19	43.99
1000000	50.23	51.57	4.38	11.47	44.31

Table 3: Sort

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
100,1000	0.04	0.05	0.02	2.00	12.50
250,2500	0.15	0.16	0.06	2.50	26.67
500,5000	0.31	0.33	0.11	2.82	29.03
1000,10000	0.70	0.76	0.23	3.04	29.29
2000,20000	1.58	1.67	0.45	3.51	32.91
2500,25000	2.06	2.15	0.55	3.75	34.47

Table 4: Shortest Path

The timing information was gathered on Sun SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during timing experiments.

Much of the data presented in the timing table is essentially self-explanatory relative to the certification trail technique and algorithms considered. However, a brief discussion of the table entries is appropriate.

The column labelled *Basic Algorithm* contains timing data which gives the execution time of the algorithm in producing the output without the generation of the certification trail. All timing data is listed in seconds.

The *First Execution* column gives the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Second Execution* column gives the execution time of the algorithm in producing the output while using the certification trail.

The *Speedup* column is the ratio of the run times of the Basic Algorithm and the Secondary Execution. One reason this figure is important is that it is possible for the two algorithms to run in different environments (different hardware, programming language, etc). A high speedup indicates that less powerful hardware or a higher level language (with associated overhead) may be sufficient for the second execution.

The *Percent Savings* column records the percentage of the execution time savings which is gained by using the certification trail method as compared to 2-version programming approach. The time required for a 2-version programming approach was estimated by doubling the time reported in the Basic algorithm. This assumes that both versions take approximately the same amount of time to execute.

In addition to the tables, the timing information for the convex hull algorithm is plotted in Figure 5. Plots for the other two examples are similar.

Examination of the data collected for the convex hull algorithm indicates that:

- The overhead in generating the certification trail is very small, less than 2% of the running time of the basic (no certification trail) algorithm.
- The second execution is very fast, achieving an order of magnitude speedup for larger input sizes. This suggests that a single "second algorithm" process could easily handle the output generated by several "first algorithm" processes running in parallel. Alternately, the high speedup would allow the second execution to be run on lower performance (and hence less expensive) hardward. Finally, the large speedup and reduced code complexity may make it possible to take advantage of a formally verifiable language (which may require significant overhead) in implementing the second algorithm.

The data for sorting indicates that the certification trail also requires very low overhead and results in a large speedup. For the shortest path problem the overhead is still very low, and the speedup, while not as dramatic as for the first two problems, is still quite respectable.

7 Comparison With Other Techniques

The certification trail approach shares similarities with other valuable fault tolerance and fault detection techniques that have been previously proposed and examined, but in each case there are significant and fundamental distinctions. These distinctions are primarily related to the generation and character of the certification trail and the manner in which the secondary algorithm uses the certification trail.

First consider the important and useful technique called N-version programming [9, 3]. When using this technique N different implementations of an algorithm are independently executed with subsequent comparison of the resulting N outputs. There is no relationship among the executions of the different versions of the algorithms other than that they all use the same input; each algorithm is executed independently without any information about the execution of the other algorithms. In marked contrast, the certification trail approach allows the primary algorithm to generate a trail of information which can be read by the secondary algorithm. The advantages of utilizing this additional information are shown in the body of this paper. In effect, N-version programming can be thought of relative to the certification trail approach as the employment of a *null trail.*

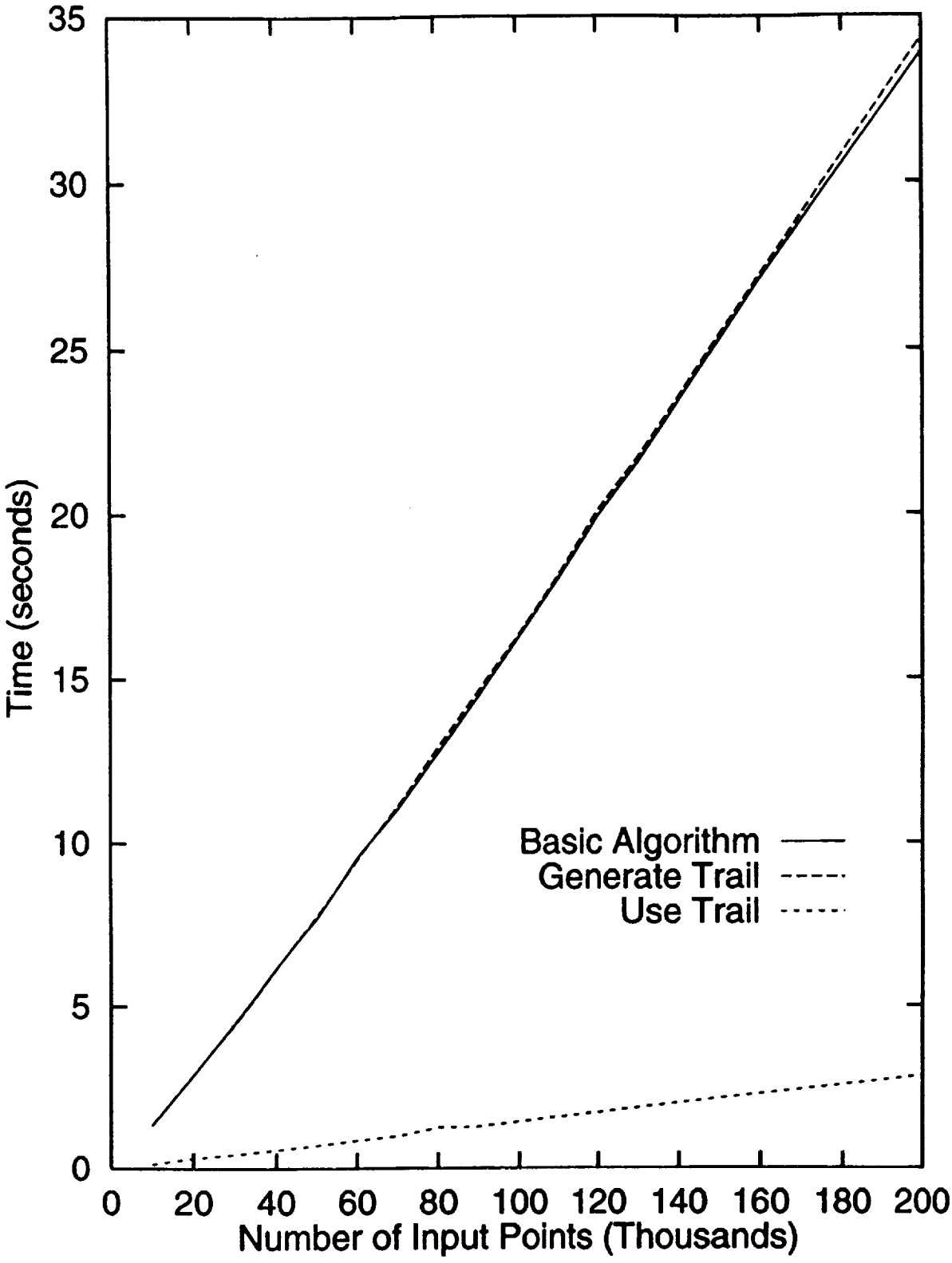


Figure 5: Convex Hull Run Times.

Another valuable technique, known as the recovery block approach [2, 18, 21], was proposed by Randell. It uses acceptance tests and alternative procedures to produce what is to be regarded as a correct output from a program. When using recovery blocks, a program is viewed as a being structured into blocks of operations, which after execution yield outputs which can be tested in some informal sense for correctness. The rigor, completeness, and nature of the acceptance test is left to the program designer, and many of the acceptance tests that have been proposed for use tend to be somewhat straightforward [2]. When using certification trails it is clearly possible to combine the second execution and the comparison test to yield a program which certifies the correctness of the output of the first execution. Unlike an acceptance test this certifier must satisfy strict formal properties of correctness. Also note that the certification trail technique emphasizes the capability of generating additional data to ease the certifying process and does not rely solely on data which would normally be computed. It should be possible to fruitfully combine the ideas of recovery blocks and certification trails.

Algorithm-based fault tolerance [15, 17, 19] uses error detecting and correcting codes for performing reliable computations with specific algorithms. This technique encodes data at a high level and algorithms are specifically designed or modified to operate on encoded data and produce encoded output data. Algorithm-based fault tolerance is distinguished from other fault tolerance techniques by three characteristics: the encoding of the data used by the algorithm; the modification of the algorithm to operate on the encoded data; and the distribution of the computation steps in the algorithm among computational units. The error detection capabilities of the algorithm-based fault tolerance approach are directly related to that of the error correction encoding utilized. The certification trail approach does not require that the data to be executed be modified nor that the fundamental operations of the algorithm be changed to account for these modifications. Instead, only a trail indicative of aspects of the algorithm's operations must be generated by the algorithm. As seen in Section 6, the production of this trail does not add significant overhead. Moreover, any combination of computational errors can be handled.

Recently, Blum and Kannan [6] have defined what they call a *program checker*. This interesting work has been followed by a burst of activity in this general area [12, 7, 25, 8, 4]. Each of these papers, however, describes work which differs significantly from the work we present. A program checker is an algorithm which checks the output of another algorithm for correctness. An early example of a program checker is the algorithm developed by Tarjan [23] which takes as input a graph and a supposed minimum spanning tree and indicates whether or not the tree actually is a minimum spanning tree.

The Blum-Kannan program checking method differs from the certification trail method in two important ways. First, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input/output specification of a problem and no assumptions are made about the method being used to solve the problem. Because of this the algorithm which is being checked is treated as a black box. It can not be altered nor can its internal status be examined and exploited. In the certification trail approach the algorithm being checked is not treated as a black box. Instead, the algorithm can be modified to generate additional information (i.e., the certification trail) which is considered to be useful in the checking/verification process. By exploiting this capability it is sometimes possible to design certification trail solutions which allow faster checking than Blum-Kannan program checkers. Of course, these faster solutions are more specialized than the Blum-Kannan checkers which are guaranteed to work for any algorithm which solves the original problem. We believe that the added speed often outweighs the disadvantage of specialization.

The second important difference concerns the number of times that the program which is being checked is executed. In the Blum-Kannan approach the program may be invoked a polynomial

number of times. In the certification trail approach the program is run only once. Thus, the overall time complexity of the checking process can be significantly larger for Blum-Kannan checkers.

A third less important difference stems from the fact that Blum-Kannan checkers are defined in a more general probabilistic context. Certification trails are currently defined only for deterministic programs and checkers. However, it is clearly possible to define them in the more general probabilistic context.

Other work has been done to extend the ideas of Blum-Kannan to give methods which allow the conversion of some programs into new programs which are self-testing and self-correcting [12, 7]. However, these methods are also based on treating programs as black boxes and thus have limitations similar to Blum-Kannan program checkers. A recent paper by Blum et al. [8] concerns checking the correctness of memories and data structures. The results described in that paper differ from our work using abstract data types in one central way. The checkers that they design are tightly constrained in memory usage. Typically, they use only $O(\log(n))$ storage to check data structures of size $O(n)$. Our results do not place space constraints on the algorithm used to certify the data structure. Without a space constraint we are able to certify abstract data types such as priority queues which are more complex than the data structures that they check, i.e., stacks and queues. Also, we are able to achieve a speed up in the checking process and they are not.

Babai, Fortnow, Levin and Szegedy [4] present methods which appear to allow remarkably fast checking, i.e., in polylogarithmic time. Their approach has some similarities to the methods we propose. Both methods modify original algorithms to yield new algorithms which output additional information. We refer to this additional information as a certification trail and they refer to this information as a *witness*. In our case we are interested in modified algorithms which have the same asymptotic time complexity as the original algorithm. Indeed, the modified algorithm should be slowed down by at most a factor of two. In [4] the modified algorithm is slowed down by more than any fixed multiplicative factor. Specifically, if the original algorithm has a time complexity of $O(T)$ then the modified algorithm has a time complexity of $O(T^{1+\epsilon})$. Note that in practice the ϵ cannot be too small because its inverse appears in the exponent of the checker time complexity. Another difference between our methods is the fact that their method requires that the input and output be encoded using an error-correcting code. The encoding process takes $O(N^{1+\epsilon})$ time for strings of length N. However, many of the checkers we have developed take only linear time so the cost of simply preparing to use their method appears to be too great in some cases. It is also necessary to decode the output after the check. Lastly, we note that Fortnow has stated that their result is currently not practical [24].

8 Generalization and Future Research Areas

The experimental timing data on certification trails indicates that this technique is of great practical value as well as of theoretical interest. Furthermore, the techniques illustrated are applicable to a wide range of problems, especially the certification of Abstract Data Types described in the shortest path example. There are many areas of interest for future exploration, a few of which are described below.

8.1 Certified Data Structure Libraries

It is apparent that the certification trail technique described for the SHORTEST-PATH program may be used for a variety of problems. Since the certification trail is produced and used by abstract data type operations, the technique may be used with any algorithm that can be implemented in terms of those abstract data types. Creating a library of such "certified data types" enables

programmers to create fault tolerant programs without having to be familiar with the certification trail technique. Object oriented programming appears to be well suited to this task.

A possible objection to this is that it provides fault detection only for the data structure implementation, since the surrounding code is simply reused. Furthermore, the data structure implementation is likely to come from library code, and hence be highly reliable. In answer to this note that:

- In many algorithms, the code using the data structure is much simpler than the code implementing the data structure.
- Although the example above illustrated reuse of using the data structures, it is certainly possible for this code to be developed separately for the first and second execution programs.
- Errors are often found even in code that has been in use for a long period of time. The added confidence of using this technique may be desirable even for library code.
- Even if the library code is highly reliable, the certification trail can be helpful in detecting errors caused by hardware problems.
- Library code may have to be tuned or even rewritten to meet for a particular application or environment, partially negating the claim of using well-tested code.

Even if fault detection is not an issue, the certification trail technique is useful during program testing and debugging. Input may be automatically generated and processed. If the output of the first and second executions differ or an error is otherwise flagged, the input set is flagged. This reduces the need to otherwise compute output for selected input and enables both more and larger sets of input to be processed. 2-version programming may be used during debugging in a similar manner, however certification trails have the advantage of reduced overhead, allowing more test cases to be run, a reduction in the hardware required for testing, or both.

8.2 Almost-concurrent execution of the certification trail

In the above discussion and examples, the certification trail programs have been executed serially, i.e., we do not run the second execution until after first execution completed. Actually, except for sorting, the two executions in the examples above can be run almost-concurrently. The "second" execution simply reads the information from the certification trail as it becomes available. The two programs will finish nearly simultaneously, the difference being in the time after the last element is read from or written to the certification trail.

8.3 Continuing after an error

A possible extension to the use of certification trails is to attempt to continue the second execution after an error is detected. Consider the shortest path example using abstract data types. In that example, the second execution used an indexed linked list that performed each operation in constant time by using the certification trail from the first execution. Suppose that an error had been detected during the second execution. Rather than simply aborting, it may be possible to continue execution. This could be done by

- Reorganizing the existing set into some other data structure (such an AVL tree, red-black tree, etc.) that allows efficient operation without a certification trail.

- Continuing to use the indexed linked list and ignoring the rest of the certification trail. Note that this would result in some operations requiring more time.
- Continuing to use the indexed linked list and attempting to use the certification trail for future operations. This may be possible if the error that occurred has sufficiently "local" effect. For example, if part of a tree structure is corrupted during the first execution, it is still possible that operations involving other parts of the tree will be performed correctly.

On a related topic, research has been done on "self-correcting" data structures in which enough redundancy is built into a data structure so that it may be reconstructed if part of it is corrupted. Using certification trails with such structures could provide an efficient detector for corruption of the data structure.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[4] Babai, L., Fortnow, L., Levin, L., and Szegedy, M., "Checking computations in polylogarithmic time, " *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 21-31, 1991.

[5] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[6] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[7] Blum, M., Luby, M., and Rubinfeld, R., "Self-testing/correcting with applications to numerical problems," *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pp. 73-83, 1990.

[8] Blum, M., Evans, W., Gemmell P., Kannan, S., and Naor, M., "Checking the correctness of memories," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science* pp. 90-99, 1991

[9] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[10] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[11] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, Sept., 1959.

[12] Gemmell, R., Lipton, R., Rubinfeld, R., Sudan, M., and Wigderson, A., "Self-testing/correcting for polynomials and for approximate functions," *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 32-42, 1991.

[13] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set", *Information Processing Letters*, pp. 132-133, 1, 1972.

[14] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[15] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

[16] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[17] Jou, J.-Y. and Abraham, J. "Fault tolerant FFT networks," *Dig. of the 1985 Fault Tolerant Computing Symposium*, pp. 338-343, IEEE Computer Society Press, June, 1985.

[18] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[19] Nair, V., and Abraham, J., "General linear codes for fault-tolerant matrix operations on processor arrays," *Dig. of the 1988 Fault Tolerant Computing Symposium*, pp. 180-185, June, 1988.

[20] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

[21] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[22] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[23] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[24] Paul Wallich, "Crunching Epsilon," *Scientific American*, pp. 22-24, Jan., 1993

[25] Andrew Chi-Chih Yao, "Coherent Functions and Program Checkers," *Proc. 22 ACM Symp. of Theory of Computing*, pp. 84-94.

Finally we discuss the work our group has performed on the design and implementation of fault injection testbeds for experimental analysis of the certification trail technique This work employs two distinct methodologies: software fault injection (modification of instruction, data, and stack segments of programs on a Sun Sparcstation ELC and on an IBM 386 PC) and hardware fault injection (control, address, and data lines of an Motorola MC68000-based target system pulsed at logical zero/one values). Our results indicate the viability of the certification trail technique. We also believe the tools we have developed provide a solid base for additional exploration.
Keywords: Software fault tolerance, certification trails, error monitoring, design diversity, data structures.

1 Introduction

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [1, 3]. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the method. We have implemented several fundamental algorithms together with versions of the algorithms which generate and utilize certification trails. Specifically, algorithms for the following problems are analyzed: huffman tree, shortest path, minimum spanning tree, sorting, and convex hull. Our results reveal many cases in which an approach using certification trails allows for significantly faster overall program execution time than a basic time redundancy approach.

We also examine algorithms for the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. For this paper we implemented and analyzed answer-validation solutions for two abstract data types. The first solution is for a simplified priority queue which allows insert, min and deletemin operations, and the second solution is for a priority queue which allows insert, min, delete and deletemin operations. In both cases, the algorithm which performs answer-validation is substantial faster than the original algorithm for computing the answers.

This paper next presents a simple probabilistic model and analysis which enables comparison between the certification-trail method and the time-

redundancy approach. The analysis shows that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the method has both a a smaller probability of error and a smaller probability of undetected error. Surprisingly, the analysis also reveals the intriguing result that the certification-trail method often can display superior performance even when the method has the same execution time or a longer execution time than the time-redundancy approach. This superior behavior stems from the typical assymetry of the execution times of the first and second executions in the certification-trail method.

The paper next discusses the work our group has performed on the design and implementation of fault injection testbeds. This work employs two distinct methodologies: software fault injection and hardware fault injection. The software fault injection tool is similar to an interactive debugger but more accurately can be considered an interactive bugger. It allows programs to be halted and faults to be injected by direct modification of the stack, data and instruction segments of a program. Output can then be captured and characterized.

The hardware fault injector is based on injecting faults into an operating microprocessor. The injection is performed by explicitly setting one or more pins of the microprocessor to logical zero and/or logical one values. The timing and duration of the pin setting is under control of a supervisory processor. The testbed also includes a multi-processor system. This system consists of three processors which are connected to one another pairwise by shared banks of dual ported memory. We plan to use this system to conduct evaluation of systems which utilize concurrent execution of algorithms using the certification-trail method.

2 Introduction to Certification Trails

To explain the essence of the certification-trail technique for software fault tolerance, we will first discuss a simpler fault-tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so-called time redundancy

[32, 52]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct. The second possibility, of undetected faults, occurs when the output of the execution is unaffected by the faults.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming [16, 12] (in this case N=2), allows for the detection of errors caused by some faults in the software in addition to those cause by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

The certification-trail technique is designed to obtain similar types of error-detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail.* This data is chosen so that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected in the data trail.

3 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Certification of Computational Results

Gregory F. Sullivan[1]
Dwight S. Wilson[2]
Gerald M. Masson[3]

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

We describe a conceptually novel and powerful technique to achieve fault detection and fault tolerance in hardware and software systems. When used for software fault detection, this new technique uses time and software redundancy and can be outlined as follows. In the initial phase, a program is run to solve a problem and store the result. In addition, this program leaves behind a trail of data which we call a *certification trail*. In the second phase, another program is run which solves the original problem again. This program, however, has access to the certification trail left by the first program. Because of the availability of the certification trail, the second phase can be performed by a less complex program and can execute more quickly. In the final phase, the two results are compared and if they agree the results are accepted as correct; otherwise an error is indicated. An essential aspect of this approach is that the second program must always generate either an error indication or a correct output even when the certification trail it receives from the first program is incorrect. We formalize the certification trail approach to fault tolerance and illustrate realizations of it by considering algorithms for the following problems: convex hull, sorting, and shortest path, We discuss cases in which the second phase can be run concurrently with the first and act as a monitor. We compare the certification trail approach to other approaches to fault tolerance.

Keywords: Software fault tolerance, error monitoring, design diversity, data structures.

1 Introduction

In this paper we describe a novel and powerful technique for achieving fault tolerance in systems. Although applicable to both hardware and software implementation, we restrict our discussion of this technique to implementation in software. To explain our technique, we will first discuss a simpler method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on a particular input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This method requires additional time, so called time redundancy [16, 22]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming [9, 3] (in this case N=2), allows for the detection of errors caused by some faults in

[1]Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.
[2]Research partially supported by NSF Grant CDA-9015667.
[3]Research partially supported by NASA Grant NSG 1442.

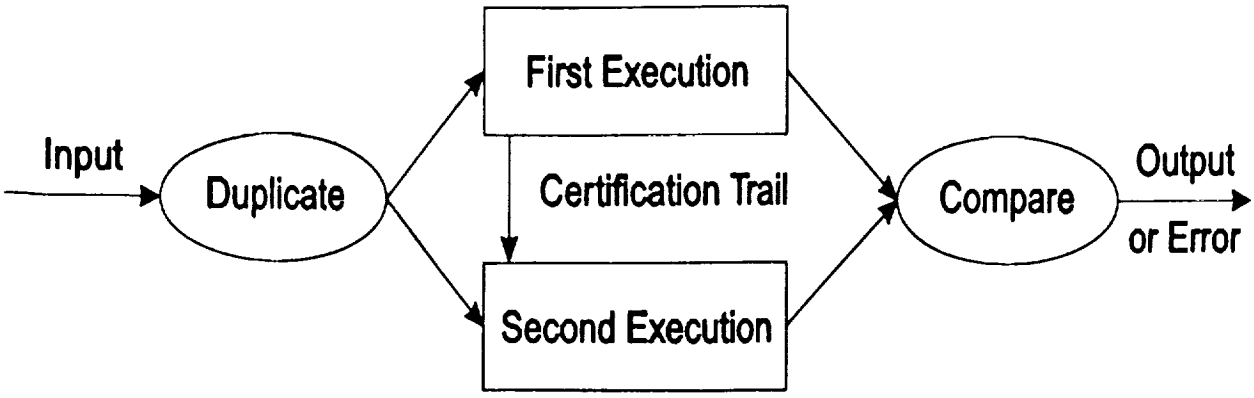


Figure 1: Certification trail method.

the software in addition to those caused by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

A significant drawback to the above approaches is the overhead required. Either extra time is required to run the algorithms serially on a single processor or extra hardware is required to run them in parallel. The technique we will describe is designed to achieve similar types of error detection capabilities while reducing the required resource overhead. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. This data is chosen to allow the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, that we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output produced by the first algorithm. Intuitively, we can regard the two executions as "adversaries." The second execution must guard against an incorrect certification trail "fooling" it into producing an incorrect output. The definitions we give below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the certification trail.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 2.1 A problem **P** is formalized as a relation, i.e., a set of ordered pairs. Let **D** be the domain (that is, the set of inputs) of the relation **P** and let **S** be the range (that is, the set of solutions) for the problem. We say an algorithm **A** solves a problem **P** iff for all $d \in \mathbf{D}$ when d is input to **A** then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 2.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. $\mathbf{T}$ is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that
$F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$
either ($F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$) or $F_2(d,t) = \text{error}$.

We also require that F_1 and F_2 be implemented so that they map elements not in their respective domains to the error symbol. The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, that the examples to be considered will indicate that this approach can also save overall execution time.

The certification trail approach also allows for the detection of faults in software. As in 2-version programming, separate teams can write the algorithms for the first and second executions. Note that the specification now must include precise information describing the generation and use of the certification trail. Because of the additional data available to the second execution, the specifications of the two phases can be very different; similarly, the two algorithms used to implement the phases can be very different. (This will be illustrated in the convex hull example to be considered later.) Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. (This will be illustrated in the shortest path example to be considered later.) When significantly different algorithms are used, the probability that both algorithms will contain or be affected by faults which generate matching errors should be reduced. When very similar algorithms are used it is sometimes possible to save programming effort by sharing program code. For example, the code implementing any data structures needed by the program might be different, while the code that uses the data structure operations would be the same. This approach is well suited for the creation of libraries of fault-tolerant data structures. While this reduces the ability to detect errors in the software it does not change the ability to detect transient hardware errors as discussed earlier. Furthermore, in situations like the above example, it is possible (perhaps even probable) that the majority of software errors will be in the data structure implementation. Thus the ability to detect software errors may not be reduced as much as first imagined.

Throughout this section we have assumed that our method is implemented with software, however, it is clearly possible to implement the method with assistance from dedicated hardware. It is also possible to generalize the basic idea we have suggested. We discuss some of these generalizations in a later section. Finally, we note that a wide variety of approaches to software fault tolerance have been proposed and we contrast our method to the most closely related ideas in a later section.

In the following two sections we illustrate the application of certification trails to three well-known and significant problems in computer science: the convex hull problem, sorting, and the shortest path problem. It should be stressed that the certification trail is not limited to these problems. Rather, these algorithms have been selected for illustrative purposes.

3 Certification Trails for Convex Hulls

The convex hull problem is a fundamental one in computational geometry. Our certification trail solution is based on a solution due to Graham [13] called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos [20]. This text also illustrates some statistical applications of convex hull computations. For simplicity in the following discussion we will assume the points are in so called general position, i.e., no three points are co-linear. It is not difficult to remove this restriction.

Definition 3.1 The *convex hull* of a set of N points, S, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique and its vertices are a subset of the points in S. It is specified by a counterclockwise sequence of its vertices.

The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects the point with minimum x-coordinate (using minimum y-coordinate to break ties), and calls it p_1. For each other point q in S we compute the slope of the line p_1q. Sort the points of S (except for p_1) by this slope (since the points are in general position, the slopes are distinct). Number these vertices $p_2, p_3, \ldots, p_N$. It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although this polygon might not be convex. It is possible to think of the algorithm as removing points from this simple polygon until it becomes convex. This code below performs this by "walking" through the vertices in order. The main FOR loop iteration adds points to the polygon under construction. After a point is added, the inner WHILE loop checks the angle formed by the addition of this point. (Note: We measure angles as follows: Given the three points q_{m-1}, q_m, p_k we measure the angle from $q_{m-1}q_m$ to q_mp_k in the clockwise direction.) If the angle is not acute (i.e., it makes the the polygon non-convex), then the angle vertex (i.e., the preceding point on the polygon) is removed. Note that this will change the preceding angle, which may now be obtuse and should be eliminated. The WHILE loop terminates when an acute angle is encountered. Figure 2 illustrates the construction of a convex hull using this algorithm. from the hull.

When the main FOR loop is complete the convex hull has been constructed.

Algorithm CONVEXHULL(S)
Input: Set of points, S, in R^2
Output: Counterclockwise sequence of points in R^2 which define convex hull of S

```
1  Let p_1 be the point with the smallest x coordinate (and smallest y to break ties)
2  For each point p (except p_1) calculate the slope of the line through p_1 and p
3  Sort the points (except p_1) from the smallest slope to the largest.
   Call them p_2, ..., p_n
4  q_1 := p_1; q_2 := p_2; q_3 := p_3; m = 3
5  FOR k = 4 to n DO
6     WHILE the angle formed by q_{m-1}, q_m, p_k is ≥ 180 degrees DO
7        m := m - 1
8     END WHILE
9     m := m + 1
10    q_m := p_k
11 END FOR
12 FOR i = 1 to m DO, OUTPUT(q_i) END FOR
```

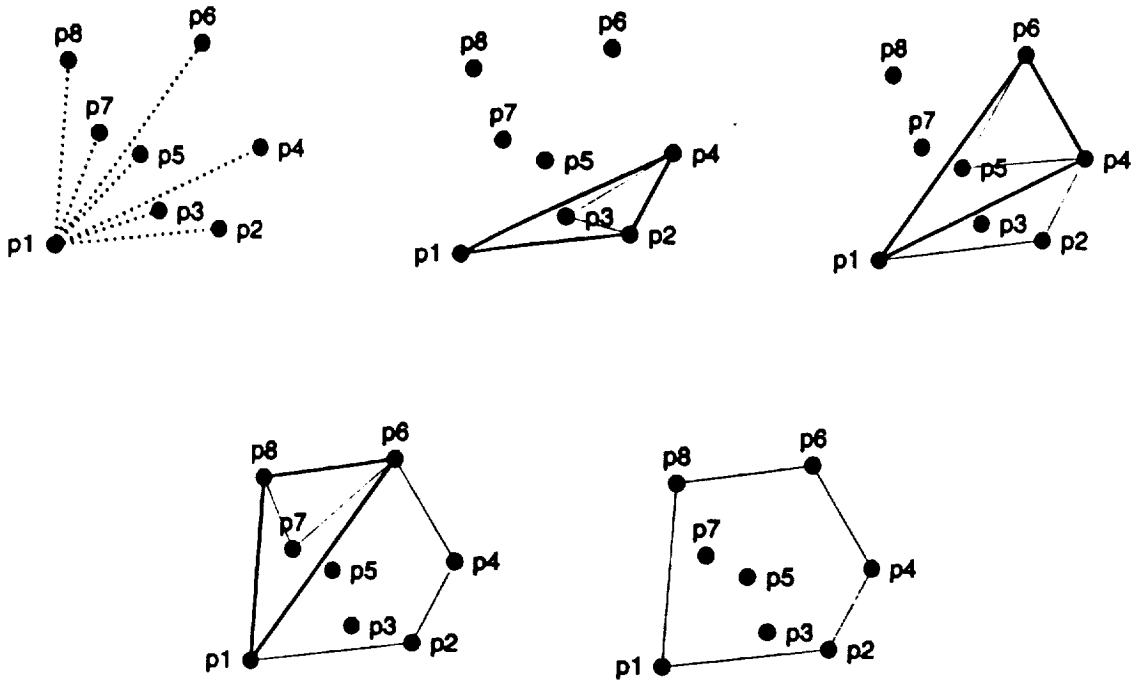


Figure 2: Convex hull example.

END CONVEXHULL

First execution: To generate a certification trail for this algorithm, we rely on the property that for each point eliminated by the WHILE loop in the code above, we can produce a triangle of points in S containing the eliminated point.

Theorem 3.2 *Let p, a, b, and c, be points in the plane such that no three are co-linear, p has the smallest x-coordinate of the four points (and the smaller y-coordinate if another other point has the same x-coordinate) $slope(\overline{pa}) < slope(\overline{pb}) < slope(\overline{pc})$. If the angle abc is obtuse (measured in the clockwise direction), then b is inside the triangle pac.*

Proof: By the ordering of the slopes, b is inside the triangular wedge determined by the rays $\vec{pa}$ and $\vec{pc}$. Note that the line segments pa and pc are in the half plain $x \geq p_x$, and in at least one case the inequality is strict, since no three points are co-linear. This implies that the angle apc (in the clockwise direction) must be greater than 180 degrees. Since the angle abc is also obtuse, both p and b must be on the same side of line $\overline{ac}$. Therefore, b is inside the triangle pac. ∎

Corollary 3.3 *During execution of CONVEXHULL, if, after adding p_k, the angle formed by q_{m-1}, q_m, p_k is obtuse (measured in the clockwise direction), then q_m is contained in the triangle p_1, q_{m-1}, p_k.*

Proof: $slope(\overline{p_1 q_{m-1}}) < slope(\overline{p_1 q_m}) < slope(\overline{p_1 p_k})$. ∎

In the first execution the code CONVEXHULL is used. The certification trail is generated by adding an output statement within the WHILE loop. Specifically, if an angle greater than 180 degrees is found in the WHILE loop test then the 4-tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The table below shows the 4-tuples of points that would be output by the algorithm when run on the example in Figure 2. The points in the table are given the same names as in Figure 2. The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Finally, the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $s_1, s_2, \ldots, s_n$ then rather than output the element in R^2 corresponding to s_i the number i is output. If point coordinates were output instead of these indices, the second execution would have to verify that the points on the trail are members of S.

Point not on convex hull	Three surrounding points
p_3	p_4, p_1, p_2
p_5	p_6, p_1, p_4
p_7	p_8, p_1, p_6

Second execution: Let the certification trail consist of a set of 4-tuples, $(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r)$ followed by the supposed convex hull, $q_1, q_2, \ldots, q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

i. That there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.

ii. That for $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.

iii. For $i \in \{1, \ldots, r\}$ that x_i lies within the triangle defined by a_i,b_i, and c_i.

iv. That for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is acute.

v. That there is a unique point among the points on the supposed convex hull which is a locally maximal point. We say a point q on the hull is a *local maximum* point if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; in fact it makes it easier to verify the first and second conditions.

Time complexity: In the first execution the sorting of the input points takes $O(n \log(n))$ time where n is the number of input points. One can show that this cost dominates and the overall complexity is $O(n \log(n))$.

It is possible to implement the second execution so that all five checks are done in $O(n)$ time. Because indices into the input data are used, the first condition can be checked by verifying that each index is used exactly once, and that all indices are between 1 and N. The second condition may checked simply by verifying that each index is between 1 and N. Checking that a point lies

within a triangle is a geometric calculation that can be done in constant time. Checking that the angle formed by three points is acute requires only constant time. The third and fourth checks can be done in $O(n)$ because the certification trail contains indices into the input data as described above. The uniqueness of the "local maximum" requires only a constant time calculation at each point, so it may checked in linear time.

Experimental timing data for this method may be found in Section 6.

3.1 Proof of correctness

We wish to prove that the algorithms above constitute a certification trail solution for the convex hull problem. Although the definition is phrased in terms of functions, not algorithms, we can simply define the functions $F_1(d)$ and $F_2(d,t)$ on particular arguments as the values computed by the associated algorithms.

Using our formal definition of certification trails, let **D** be the set of all finite planar point sets T. Let **S** be the set of convex polygons, with vertices in counterclockwise order (the restriction to counterclockwise ordering makes the convex hull unique). Then the problem we are considering is $HULL : \mathbf{D} \rightarrow \mathbf{S}$ where $HULL(T)$ is the polygon in **S** that forms the convex hull of T.

The description of the algorithms above defines functions F_1 and F_2. We must show that both conditions of Definition 2.2 hold. The following two lemmas, which we state without proof, are required.

Lemma 3.4 *Let P be a polygon on n points $p_1, p_2, \ldots, p_n$. P is a convex polygon iff P is simple and each angle $p_i p_j p_k$ is less than or equal to 180 degrees, where i is in $1, 2, \ldots n$, $j = (i+1) \bmod n$, and $k = (i+2) \bmod n$.*

Lemma 3.5 *If P is a non-simple polygon, then either P has more than one local maxima, or the interior angle at some vertex is greater than 180 degrees.*

Theorem 3.6 *$F_1(d)$ and $F_2(d,t)$, as defined above, constitute a certification trail solution for the problem HULL.*

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: Recall that the first condition is: for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. Intuitively, this means that if both executions perform correctly, then they will both output the convex hull of the input, which is unique. Note that generation of the certification trail does not affect the output of the Graham Scan algorithm. Thus the condition on $F_1(d)$ is satisfied by the correctness of the Graham Scan algorithm, the proof of which is well known [20]. To show that $F_2(d,t) = s$, note that a copy of s is contained on the trail t. Our description of $F_2(d,t)$ states that s is output unless one of the five checks above fails. It is trivial to verify that the first three of these checks must be satisfied. The fourth check cannot fail, since the polygon described by s is convex (because $(d,s) \in \mathbf{P}$). Similarly, if the fifth check fails, then the polygon described by s has two local maxima, and this is not possible for a convex polygon.

Part 2: The second condition is: for all $d \in \mathbf{D}$ all $t \in \mathbf{T}$ either ($F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$) or $F_2(d,t) =$ error. Intuitively, this means that given an input and arbitrary trail, $F_2(d,t)$ produces a solution to the problem or flags an error. Our definition of $F_2(d,t)$ states that the polygon Q stored on the trail is output unless one of the five checks fails. We must therefore demonstrate that if all five checks succeed, then Q is the convex hull of the input points d. Let H be the convex hull of the points d. The first condition guarantees that every point in d is classified as a hull point or an

interior point. The second condition guarantees that the triangles used to identify interior points are formed from input points, and the third check verifies that the interior points are indeed inside their respective triangles. Note that we do not attempt to verify that the triangles on the trail are the ones that would be produced by $F_1(d)$. In general, for a given interior point, there may be several triangles of input points in which it is contained. Together, the first three conditions imply that all points in H are also in Q, since it is impossible for a hull point to be contained in a triangle. Note that these three checks do not exclude the possibility that interior points are present in Q, nor do they guarantee that the ordering of the hull points in Q is correct. The final two checks will accomplish this. If the last two checks are satisfied, Lemma 3.5 states that Q is simple, and therefore it must be convex by Lemma 3.4.

Thus, Q is a convex polygon whose vertex set is a superset of the vertices of H, i.e., H is contained in Q. This implies that no other point from the input set may be a vertex of Q, since any input point that is not a hull point is interior to H and therefore interior to Q. Finally, it is clear that the ordering of the vertices of Q and H must be the same (although there might appear to be two possible orderings, clockwise and counterclockwise, a clockwise ordering will fail the fourth check). Therefore if all five checks succeed, then the output of $F_2(d,t)$ will be the convex hull of d.

This demonstrates that the algorithms described meet the conditions of Definition 2.2, and are therefore a certification trail solution to the convex hull problem. ∎

3.2 Other convex hull algorithms

It is possible to use this technique to provide certification trails for other convex hull algorithms. The key is that for each non-hull point p we must find a triangle of input points (not necessarily hull points), containing p. For some convex hull algorithms, a containing triangle is available directly or can be easily computed when it is determined that a particular point is not on the hull. However, this is not true of all convex hull algorithms. If, however, we allow extra overhead during the first execution we may apply this technique to any planar convex hull algorithm, provided that the output is a polygon and not merely an unordered list of hull vertices.

Let $H = q_1, q_2, q_3 \ldots, q_h$ be the convex hull of a set of n points. We label the points so that q_1 is the point with smallest abscissae (and smallest ordinate in case of a tie). Since H is convex, the remaining points occur in sorted angular order around q_1. Now for each non-hull point p, we may determine which triangle $p_1 p_i p_{i+1}$ it lies in with a binary search. Thus we may determine containing triangles for the non-hull points in $O(n \log h)$ time. Under several distributions the number of hull points is much smaller than the number of input points [20] so this overhead will often be quite small.

4 Sorting

Sorting is one of the most important basic problems in computer science. There is a massive body of literature discussing sorting and a significant fraction of computer time is spent performing sort operations. We will see how the certification trail approach may be applied to this problem. Assume that a particular sorting algorithm takes as input an array of n elements and outputs an array of n elements. The algorithm is supposed to place the data into non-decreasing order.

Note that it may not appear necessary to use a certification trail for this problem. It might seem that all that is required is to verify that the output is in non-decreasing order. Unfortunately, this is not sufficient and we must also verify that the output consists of the same elements as the input. A certification trail is required to perform this check efficiently.

The information placed on the trail is a permutation relating the input and output arrays. This permutation is created by adding an Item Number field to the elements being sorted, such that the i-th element is labelled with item number i. After sorting, the permutation is obtained by reading the Item Numbers from the elements in their new order.

The second algorithm reads the permutation from the trail, uses it to rearrange the input elements in linear time, and checks that they are now in sorted order. Additionally, it is necessary to check that the the information on the certification trail actually is a permutation of n elements, i.e., each number from 1 to n occurs exactly once. Should any of these checks fail, the second algorithm outputs "error", otherwise it outputs the sorted elements.

Note that the certification trail given for sorting is quite different than that given for the convex hull problem. In the latter case, the certification trail was constructed for a particular algorithm, and the code executing that algorithm modified to produce the trail. In this case, the sorting algorithm is not changed. Instead the data being sorted is modified by a preprocessing step, and the necessary information extracted by a postprocessing step. Thus this technique may be implemented as a "wrapper" around existing sort routines, no matter which algorithm is implemented.

Experimental data is presented in Section 6.

4.1 Proof of correctness

For concreteness we consider only the sorting of integers, though the proof does not depend on this condition.

Definition 4.1 Let $\mathbf{D}$ consist of all finite sequences of integers. Let $\mathbf{S}$ consist of all finite non-decreasing sequences of integers. Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be the sorting problem, i.e., $(d, s) \in \mathbf{P}$ iff s is a permutation of d (by definition of $\mathbf{S}$, s is a non-decreasing sequence). Note that for every $d \in \mathbf{D}$, there is a unique $s \in \mathbf{S}$ such that $(d, s) \in \mathbf{P}$. Let $\mathbf{T}$ consist of finite sequences of integers. For x a member of any of the sets $\mathbf{D}, \mathbf{S}$, or $\mathbf{T}$, we will also denote the sequence of integers by $x_1, x_2, ..., x_N$.

Definition 4.2 The function $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ is defined as follows. Given an input sequence d of N integers, $F_1(d) = (s, t)$ where s is the unique element of $\mathbf{S}$ such that, $(d, s) \in \mathbf{P}$ and t is a permutation of 1,2,3,...,N s.t., $s_i = d_{t_i}$ for all $i = 1, 2, ...N$. Note that unless d consists of N distinct integers, there will be more than one possible t. The t produced by $F_1(d)$ may be chosen arbitrarily. Since for every $d \in \mathbf{D}$, there exists a unique $s \in \mathbf{S}$ with $(d, s) \in \mathbf{P}$, the function F_1 is well defined.

Definition 4.3 The function $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$ is defined as follows. $F_2(d, t) = d_{t_1}, d_{t_2}, ..., d_{t_N}$ (where d consists of N integers) iff

i. t contains at least N integers.

ii. The first N integers of t are a permutation of $\{1, 2, ...N\}$.

iii. $d_{t_i} \leq d_{t_{i+1}}$ for $i = 1, 2, ..., N - 1$.

Otherwise, $F_2(d, t) =$ error. Note that though t may contain more than N integers, $F_2(d, t)$ depends only on the first N.

The definitions of the functions F_1 and and F_2 correspond to the informal descriptions of the sorting algorithms given in the text above.

Theorem 4.4 *F_1 and F_2 are a certification trail solution to the sorting problem* $\mathbf{P}$.

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: We must prove that for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. If $F_1(d) = (s,t)$, then by definition $(d,s) \in \mathbf{P}$. We must show that $F_2(d,t) = s$. t is a permutation of $\{1,2,...,N\}$, so the first two conditions of Definition 4.3 are satisfied. Furthermore, by Definition 4.2, $d_{t_i} = s_i$ for $i = 1,2,...N$. Since $s \in \mathbf{S}$, it is a nondecreasing sequence, and thus the third condition of Definition 4.3 is satisfied. Therefore $F_2(d,t) = s$.

Part 2: We must show that for all $d \in \mathbf{D}$ and all $t \in \mathbf{T}$ either ($F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$) or $F_2(d,t) =$ error. Pick $d \in \mathbf{D}$ with length N. Pick $t \in \mathbf{T}$. The interesting case is when t is a permutation of $\{1,2,...,N\}$. If not, then either the first N integers of t are not such a permutation, in which case $F_2(d,t) =$ error. We may ignore the possibility that t consists of such a permutation followed by more integers, since F_2 depends only on the first N integers of t.

Examine the sequence $d_{t_1}, d_{t_2},,, d_{t_N}$. If there is an i such that $d_{t_i} > d_{t_{i+1}}$ then the third condition of Definition 4.3 is violated so $F_2(d,t) =$ error. Otherwise $F_2(d,t) = d_{t_1}, d_{t_2}, ..., d_{t_N}$. Furthermore, this is a non-decreasing sequence, so it must be in $\mathbf{S}$. Finally, since this sequence is a permutation of d, $(d, F_2(d,t)) \in \mathbf{P}$.

Therefore, both conditions of Definition 2.2 are satisfied, so F_1 and F_2 constitute a certification trail solution to sorting. ∎

Note that we defined $\mathbf{T}$ as the set of all finite sequences of integers. We could have instead defined $\mathbf{T}$ as the set of permutations of $\{1,2,...N\}$ for all positive N. This would make the function F_2 "simpler", in that it doesn't have to verify that that certification trail consists of a permutation (it would, however, have to verify that it consists of a permutation of the correct size). In this case, checking that the trail t is indeed a permuation (i.e., actually in its domain) would be left to the implementation of the function.

5 Certification Trails for Shortest Paths

This classic problem has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [11] as explicated in [10]. First we require some preliminary definitions.

Definition 5.1 A *graph* $G = (V,E)$ consists of a *vertex* set V and an *edge* set E. An edge is an unordered pair of distinct vertices which we notate with the following style: $[v,w]$ and we say v is adjacent to w. A *path* in a graph from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $[v_i, v_{i+1}]$ is an edge for $i \in \{1,\ldots,k-1\}$. Let w be a real function defined on E. The *length* of a path from v_1 to v_k is the sum of $w([v_i, v_{i+1}])$ for each edge $[v_i, v_{i+1}]$ in the path.

Let $G = (V,E)$ be a graph and let w be a positive rational valued function defined on E. Given a vertex v_1 in V, find a set of shortest paths from v_1 to each other vertex in V. Note that since w is positive on all edges, a shortest path must exist between any two vertices, though it need not be unique.

Before we discuss the algorithm we must describe the properties of the principal data structure that are required. Since many different data structures can be used to implement the algorithm, we initially describe abstractly the data that can be stored by the data structure and the operations that can be used to manipulate this data. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the *item number* and the second element is called the *item value* or just *value*. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for

multiple ordered pairs to have the same item value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, x is a value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i,x) < (i',x')$ iff $x < x'$ or $(x = x'$ and $i < i')$. Our data structure should support a subset of the following operations.

member(i,h) returns a boolean value of true if h contains an ordered pair with item number i, otherwise returns false.

insert(i,x,h) adds the ordered pair (i,x) to the set h.

delete(i,h) deletes the unique ordered pair with item number i from h.

changekey(i,x,h) is executed only when there is an ordered pair with item number i in h. This pair is replaced by (i,x).

deletemin(h) returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If h is the empty set then the token "empty" is returned.

predecessor(i,h) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned.

A description such as the one above describes an *abstract data type*. There may be several possible implementations for a particular ADT. In our solution, different ADT implementations will be used for the two executions. The first implementation will produce a certification trail allowing the second implementation to be simpler and to perform ADT operations more quickly.

Aside from the implementation of the abstract data type, both of our algorithms are the same. Pidgin code for this algorithm appears below. Figure 3 illustrates the execution of the algorithm on a sample graph. Table 1 records the data structure operations performed when the algorithm is run on the sample graph. The first column gives the operations, with the parameter h omitted to reduce clutter. Member operations are also omitted from the table. The second column gives contents of h after the execution of each instruction. The third column records the order pair deleted by deletemin operations. The fourth column records the information (if any) output to the certification trail by this operation.

This certification trail is created by modifying the insert(i,x,h) and changekey(i,x,h) operations performed during the first execution. The modified instructions perform the same operations described above and in addition output the following information to the certification trail.

insert(i,x,h) Output the item number of the predecessor of (i,x) (as defined above) to the trail. If there is no predecessor, output the token "smallest". Note that depending on the data structure implementation, the predecessor may already be computed during insertion or may require a separate call to the predecessor(i,h) operation.

changekey(i,x,h) Output the predecessor of the ordered pair (i,x) (i.e., pair resulting from the change) to the trail. If there is no predecessor, output the token "smallest" to the trail.

We shall see that this information allows a faster and simpler data structure implementation to be used for our second algorithm.

The algorithm proceeds by maintaining a set S of vertices for which shortest path lengths are known, and a "frontier" set F of vertices adjacent to members of S along with the best known path

length from v_1. At each step, we find the vertex v in F with smallest known path length and place it in S, F is then updated by examining the neighbors of v. New vertices may be added to F or a shorter path (passing through v) may be found to existing vertices in F.

To efficiently find the vertex to add to S, the algorithm uses the data structure operations described above. As soon as a vertex v is adjacent to some vertex u in S, it is inserted in the set F. The value for v is the shortest known path to v, which is the value of u (shortest path to u) plus the weight of edge vw. The array element prefer(v) is used to keep track of this "best" edge connecting v to S. As the tree grows, information is updated by operations such as insert(i, x, h) and changekey(i, x, h). The deletemin(h) operation is used to select the next vertex to add to the span of the current tree. Note, the algorithm does not explicitly store paths. Implicitly, however, if (v, x) is returned by deletemin, then prefer(v) indicates the predecessor of v on the shortest path from v_1.

Algorithm SHORTEST-PATH(G,v_1,weight)
Input: Connected graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ with edge weights.
Output: Lengths of shortest paths from v_1 to all other vertices.

```
1  FOR ALL u ∈ V, u) := ∞ END FOR
2  v1) := 0
3  F := v1;
4  WHILE F ≠ ∅ DO
5   (v, k) := deletemin(F)
6   FOR EACH [v, w] ∈ E DO
7    IF v) + weight([v, w]) < w) THEN
8     w) := v) + weight([v, w]); prefer(w) := v
9     IF member(w, F) THEN changekey(w, w), F)
10     ELSE insert(w, w), F) END IF
11   END IF
12  END FOR
13 END WHILE
14 FOR ALL u ∈ V − {v1}, OUTPUT(u)) END FOR
END SHORTEST-PATH
```

Note that this code may be easily modified to output the shortest paths as well as their lengths.

First execution: In this execution the SHORTEST-PATH code is used and the abstract data type is implemented with a balanced search tree such as an AVL tree [1], a red-black tree [14], or a b-tree [5]. In addition, an array indexed from 1 to n is used. Each element of this array contains two fields, *InSet*, a boolean, and *Value*, storing the same type as the value used in the ordered pairs. Initially, *InSet* is false for all array elements. The balanced search tree stores the ordered pairs in h and is based on the total order described earlier. For each item number i, the *InSet* field of the i-th array element is true if and only if there is a pair with item number i in the set. The *Value* field of the i-th array element stores the value of the pair with item number i, if there is one in the set. It is undefined if there is no such pair in the set. This array allows rapid execution of operations such as member(i, h) and delete(i, h).

Second execution: This execution also uses the SHORTEST-PATH code, however, a different data structure is used to implement the ADT. We call this data structure an *indexed linked list* and it is depicted in Figure 5. It consists of an array and a doubly linked list. The array is indexed from 0 to n and contains pointers to the elements of the linked list. Except for the first element,

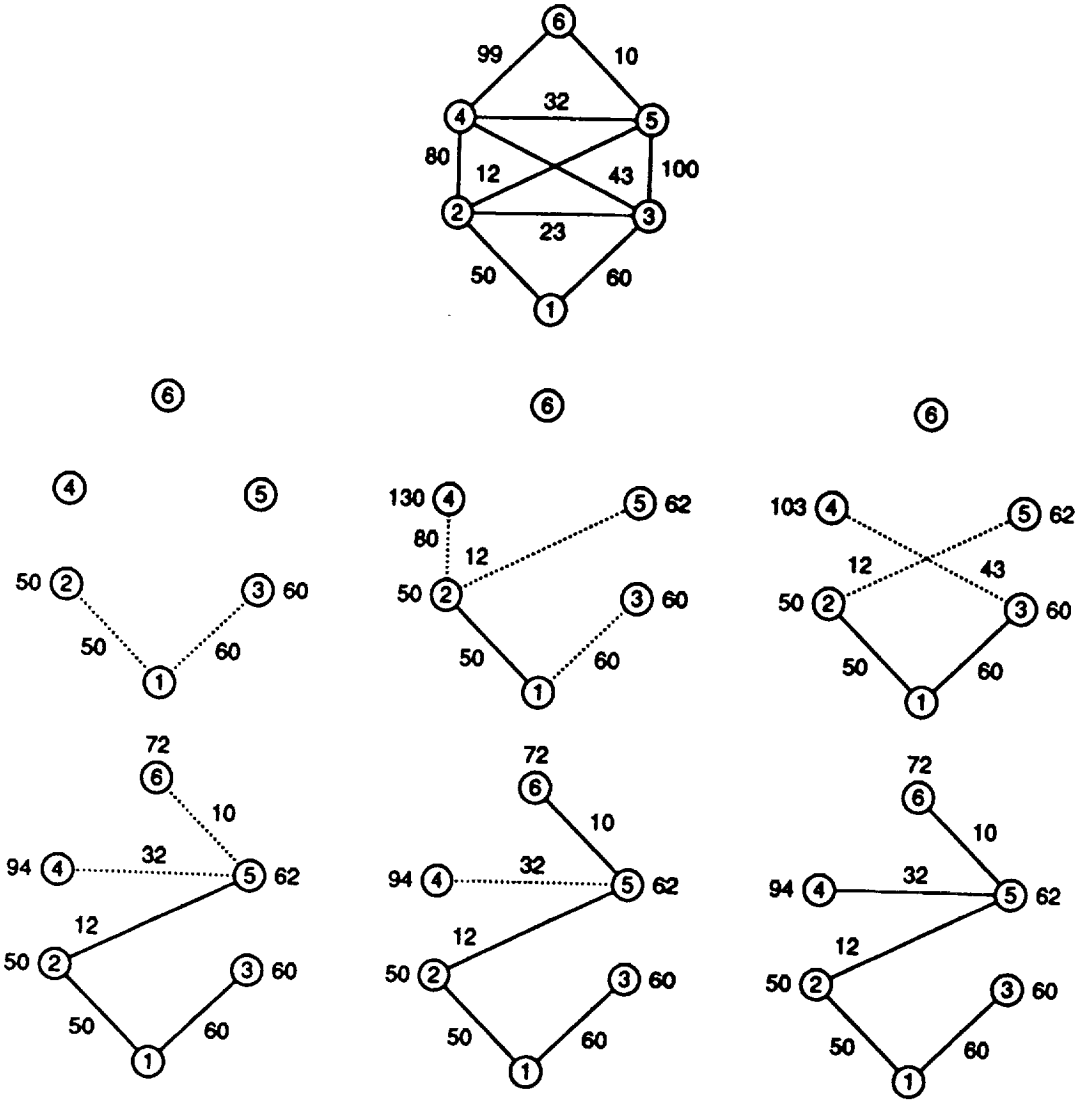


Figure 3: Shortest path example.

Operation	Set of Ordered Pairs	Delete	Trail
insert(2,50)	(2,50)		smallest
insert(3,60)	(2,50),(3,60)		2
deletemin	(3,60)	(2,50)	
insert(4,130)	(3,60),(4,130)		3
insert(5,62)	(3,60),(5,62),(4,130)		3
deletemin	(5,62),(4,130)	(3,60)	
changekey(4,103)	(5,62),(4,103)		5
deletemin	(4,130)	(5,62)	
changekey(4,94)	(4,94)		smallest
insert(6,72)	(6,72),(4,94)		smallest
deletemin	(4,94)	(6,72)	
deletemin		(4,94)	
deletemin		empty	

Table 1: Example of operations and trail.

each element in the list contains a data field storing an ordered pair. The first element stores a special ordered pair $(0, \text{"}smallest\text{"})$ which is guaranteed to compare less than any other ordered pair. The list is maintained in sorted order based on the total ordering defined above for ordered pairs. This list represents the contents of the set h. The i-th element of the array points to the node containing the ordered pair with item number i, if such an element is present in h. Otherwise the pointer is nil. The 0-th element of the array points to the node containing $(0, \text{"}smallest\text{"})$ Initially, all pointers are nil except for the 0-th one. Using an ordered list allows us to perform deletemin(h) operations quickly. The array provides rapid random access to the elements. We now describe the implementation of the data structure operations.

insert(i, x, h) Read the next value from the certification trail. This value, call it j, is the item number of the ordered pair that will be the predecessor of (i, x) after it is inserted. To insert this element, we follow the j-th array pointer to the list node containing the pair (j, y). There is one special case, if "smallest" is read from the trail rather than an item number, we follow the 0-th pointer. A new node is allocated and inserted into the list just after the node containing (j, y). The data field of this node is set to (i, x). Finally, the i-th pointer is set to point to the new node. Figure 5 shows the insertion of (5,62) into the data structure, given that the next item on the certification trail is 3. When the insert(i, x, h) operation is performed, some checks must be conducted:

i. The i-th array element must be nil before the operation is performed.

ii. The value j read from the trail must either be "smallest" or be between 1 and n, i.e., it must be a valid item number.

iii. The j-th array element must not be nil before the operation is performed.

iv. The sorted order of the pairs stored in the linked list must be maintained. That is, if the j-th pointer points to (j, y) and its successor before the insertion (ignoring the

special case when (j, y) is the last element of the list) is (j', y'), then we must have $(j, y) < (i, x) < (j', y')$.

If any of these checks fails, then the execution halts and "error" is output.

delete(i, h) If the i-th pointer is nil, halt execution and output "error". Otherwise follow the i-th pointer to find the list node containing (i, x). This node is removed from the list. Note that since the list is doubly linked, this is a constant time operation. The i-th pointer is then set to nil. The only condition that must be checked is that the i-th pointer is not nil before the deletion

changekey(i, x, h) To perform this operation, it suffices to perform delete(i, h) followed by insert(i, x, h). The next item for the certification is read when the insert(i, x, h) operation is performed. If any of the conditions required by either of these operations fails, then execution halts and "error" is output.

deletemin(h) The 0-th array pointer is traversed to the list head (which contains $(0, \text{"}smallest\text{"})$). The pointer to the next node in the list is followed. If there is no next node then "empty" is returned. Otherwise, let (i, x) be the pair stored in that node. We remove the node from the list, set the i-th array element to nil, and return (i, x).

member(i, h) The i-th array pointer is examined. "False" is returned if it is nil, otherwise "true" is returned.

predecessor(i, h) This operation is not used during the second execution of SHORTEST-PATH, but is described for completeness. Follow the i-th pointer to the node containing the pair (i, x). Follow the pointer from that node to the node preceding it on the list (note that this node will always exist). If this is the special node $(0, \text{"}smallest\text{"})$, return "smallest", otherwise return the item number of the pair stored in this list.

There are two variations to this scheme that are worth noting. First, we could implement a singly linked list rather than a doubly linked list. This eliminates the overhead of maintaining the extra pointer. Note, however, that operations such as delete(i, h) require access to predecessors in order to update the list quickly. This can be provided by modifying the operations delete(i, h), changekey(i, x, h), and predecessor(i, h) so that they output predecessor information to the trail.

The other variation also uses a singly linked list but removes the need for extra certification trail information for delete(i, h) and changekey(i, x, h) operations. It uses the technique of marking a list node for deletion rather than removing them from the list node immediately (the appropriate pointer in the array is still set to nil immediately). When performing other operations, we check for and remove any marked nodes immediately following nodes visited. The total running time is still linear, though insert operations are no longer constant time operations.

Time complexity: In the first execution each data structure operation can be performed in $O(\log(n))$ time where $|V| = n$. There are at most $O(m)$ such operations and $O(m)$ additional time overhead where $|E| = m$. Thus, the first execution can be performed in $O(m \log(n))$ In addition, it provides us with a relatively simple and illustrative example of the use of a certification trail.

In the second execution each data structure operation can be performed in $O(1)$. There are still at most $O(m)$ such operations and $O(m)$ additional time overhead. Hence, the second execution can be performed in $O(m)$ time, i.e., linear time.

Section 6 contains results of timing experiments with this technique.

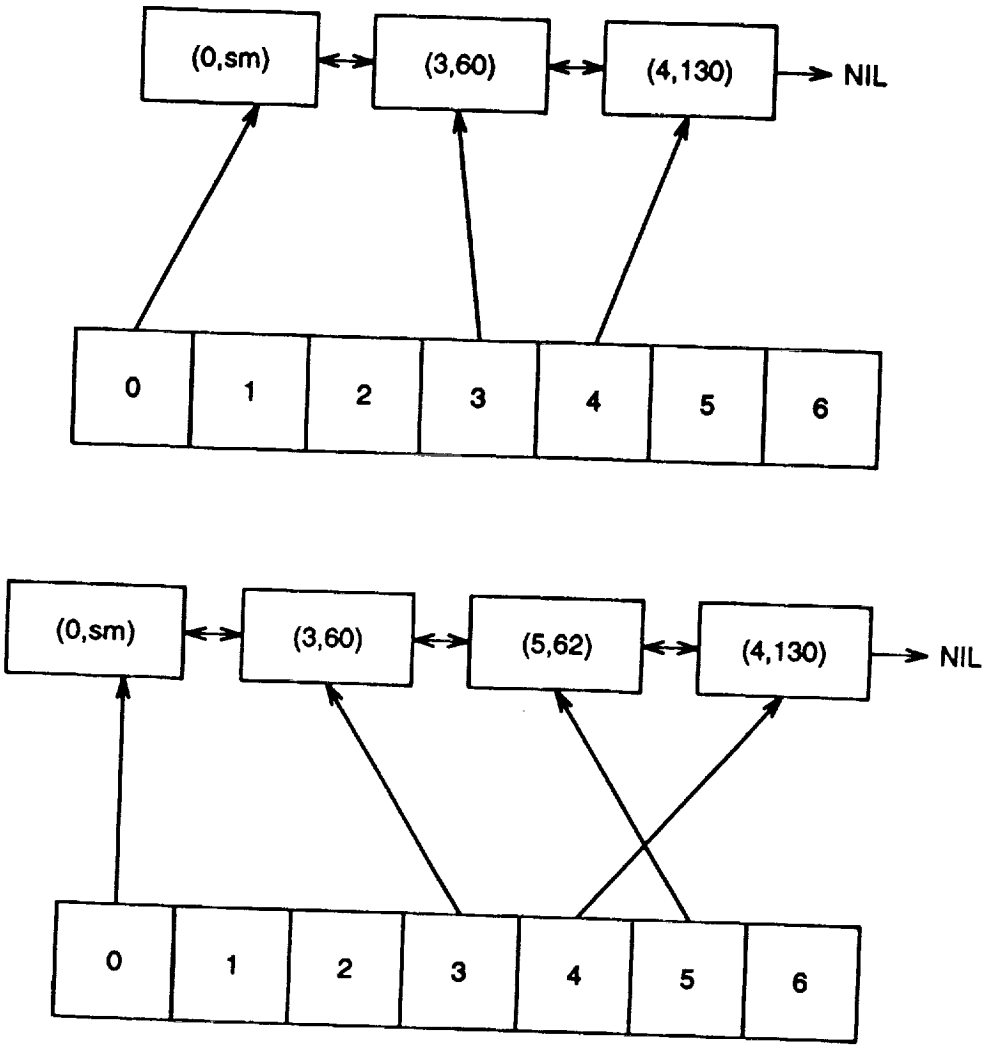


Figure 4: Example of the indexed linked list before and after inserting (5,62)

5.1 Proof of correctness

We wish to prove that the two algorithms given above constitute a certification trail solution to the SHORTEST-PATH problem, i.e., that the functions $F_1(d)$ and $F_2(d,t)$ defined by these algorithms satisfy Definition 2.2. First, we consider the problem of evaluating a sequence of the above data structure operations.

Definition 5.2 Let **D** be the set of finite sequences of the data structure operations defined above. Let **S** be the set of finite sequences of answers to data structure operations. Let **P** be the relation (d,s) where $d \in \mathbf{D}$ and $s \in \mathbf{S}$, and s is the sequence of answers resulting from executing the operations d starting with the empty set.

Note that we are examining all finite sequences of data structure operations, not just "legal" ones. That is, may attempt to perform an insertion with an item number already in use, attempt to perform deletion on an item number not being used, etc. We assume that if one of these "illegal" operations is attempted, the operation will output "error" and terminate processing. Thus, we can define the answer sequences for these "illegal" sequences.

Definition 5.3 Let $F_1(d)$ be defined by the result of executing the operations on any of the standard data structures described above, with the insert(i,x,h) and changekey(i,x,h) operations modified to output trail information. Let $F_2(d,t)$ be defined by the result of executing the operations using the indexed linked list implementation described above.

Theorem 5.4 *$F_1(d)$ and $F_2(d,t)$ meet the conditions of Definition 2.2 (that is, $F_1(d)$ and $F_2(d,t)$ constitute a certification trail solution for* **P***).*

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: The first condition we must verify is that for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. Let $(s,t) = F_1(d)$. The modifications of the data structure operations that produce trail output do not affect how the data structure is maintained. Proofs of correctness for the standard data structures are well known, so we may assume $(d,s) \in \mathbf{P}$. We must demonstrate that $F_2(d,t) = s$.

This may be proven by showing that after each operation that modifies the set h, the elements stored in the indexed linked list (our implementation) correspond to the elements in the set h (the abstract definition). We must also demonstrate that if this relationship is maintained, then correct output is generated by operations that generate output.

To demonstrate this, we show that each operation maintains the following invariants.

i. If the pair (i,x) is in $h \cup (0,$ "*smallest*"$)$, then the i-th pointer in the array of pointers points to the list node containing (i,x).

ii. If, for some i, there is no pair in h with item number i then the i-th pointer is nil.

iii. The list nodes are in ascending order.

iv. Every list node is pointed to by some pointer in the array. (Together with the first condition, this implies that it is pointed to by exactly one pointer from the array).

The first two conditions assert that the indexed linked list and the set h contain the same elements (ignoring the special list head element in the linked list). The last two invariants allow us to demonstrate that the linked list operations function correctly.

Clearly each of these conditions is true before the first operation is performed (the set of pairs is empty, all pointers except the 0-th are nil, and $(0, \text{"}smallest\text{"})$ is the only list node).

Assume that the above conditions are satisfied after the first k operations, and that the output generated by any of the first k operations is correct. We claim that the invariants will will remain satisfied after the $(k+1)$-st operation, and that if the $(k+1)$-st operation generates output, it will be correct. Let $s(k+1)$ denote the output produced by the $(k+1)$-st operation (where $F_1(d) = (s, t)$).

Consider each possible operation. For brevity, we omit details for "illegal" operations, i.e., those that violate the precondition of the operation. Similarly, we omit details of the special case of "smallest" being read from the trail.

insert(i, x, h) The trail t contains the item number j of the predecessor of (i, x). Call the predecessor (j, y). By assumption, the i-th pointer is nil before the insert. If not, this operation outputs "error" and execution halts. Since the indexed linked list correctly represents h at this point, this agrees with the result returned by $F_1(d)$, i.e., $s(k+1) = \text{"}error\text{"}$. After the insertion is performed, the i-th pointer is set to the new node containing (i, x), so the first condition is satisfied. No other nodes are added to the list, so the second condition will remain true. The third condition is satisfied since (j, y) is now the immediate predecessor of (i, x). Since no other pointer in the array has been changed, the fourth condition is still true.

delete(i, h) This operation sets the i-th pointer to nil, and removes the node containing (i, x) from the list. This satisfies the second invariant. Deleting a node cannot violate the third invariant. Since no other nodes are removed and no other pointers are changed, the first and fourth invariants remain satisfied.

deletemin(h) By assumption, the nodes are currently in ascending order. Thus, the minimum element in h must correspond to the node following the special list head node, call the pair it contains (i, x). This pair is the correct output for this operation. As with delete, the above four conditions remain true after this node is removed and the i-th pointer set to nil.

changekey(i, x, h) We have implemented changekey(i, x, h) as an insertion followed by a deletion. Since both of those preserve the invariants, changekey(i, x, h) must do so as well.

member(i, h) By assumption, the indexed linked list correctly represents h before this operation, so the output of this operation will be correct. Since this operation does not change the set or the indexed linked list, the invariants remain satisfied.

predecessor(i, h) By assumption, the indexed link list correctly represents h, and furthermore it is currently in sorted order. Thus, the list element preceding the node containing (i, x) is the predecessor. Since this operation changes neither h nor the indexed linked list, the invariants remain satisfied.

This demonstrates that the first condition of Definition 2.2 is satisfied.

Part 2: The second condition is for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$ either ($F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$) or $F_2(d, t) = \text{error}$. Intuitively, this states that if $F_2(d, t)$ is passed an arbitrary trail, it either outputs a correct answer, or it outputs "error". We prove an even stronger condition. Let $t_{correct}$ be the trail returned by $F_1(d)$, i.e., $F_1(d) = (s, t_{correct})$. Then either $t_{correct}$ is a prefix of t, or $F_2(d, t) = \text{error}$.

If $t_{correct}$ is a prefix of t, then we are done. The algorithm describing $F_2(d, t)$ does not examine any part of the trail after $t_{correct}$, so $F_2(d, t) = s$.

If $t_{correct}$ is not a prefix of t, let p be the position at which they first differ. Let O be the number of the operation that uses the trail data at p. Then operation O is either an insert(i, x, h) or changekey(i, x, h) operation. If it is an insert operation, then $t_{correct}$ contains the item number of the predecessor of (i, x). Since t contains a different value, call it j, at this location, the insert(i, x, h) operation will fail one of it's three checks. Either j will not be valid item number, or the j-th pointer will be nil, or the pair (j, y) will not be the predecessor of (i, x). The argument for the changekey(i, x, h) operation is essentially the same.

Thus, the second condition is satisfied.

Therefore, $F_1(d)$ and $F_2(d, t)$ are a certification trail solution to **P**, the problem of evaluating data structure operations. ∎

Definition 5.5 Let **D** be the set of finite graphs $G = (V, E)$ with edge weights consisting of positive integers. Assume the indices are numbered 1 through n. Let **S** be the set of finite ordered tuples of positive integers. Let **P** be the relation that associates each graph with the tuple consisting of the minimum path lengths to each vertex. Let $SP_1(d)$ be the function defined by the SHORTEST-PATH algorithm with the data structure defined for the first execution. Let $SP_2(d, t)$ be the function defined by the SHORTEST-PATH algorithm using the indexed linked list implementation.

Corollary 5.6 *$SP_1(d)$ and $SP_2(d, t)$ constitute a certification trail solution for* **P**.

Proof: If $SP_1(d) = (s, t)$, then the correctness of Dijkstra's algorithm implies that $(d, s) \in$ **P**. The algorithms that compute $SP_1(d)$ and $SP_2(d, t)$ are the same except for data structure implementation. Theorem 5.4 implies that if these algorithms generate the same data structure operations, then the same sequence of answers will be generated. Thus, to demonstrate that $SP_2(d, t) = s$, it must be shown that the same sequence of data structure operations is generated by both algorithms. Examination of SHORTEST-PATH indicates that the k-th data structure operation to be performed is dependent only on the input and the result of previous data structure operations. For example, at line 9, either an insert(i, x, h) or a changekey(i, x, h) is performed, depending on the result of a member(i, h) operation. The input graph d is identical for both algorithms, thus the first data structure operation performed must be the same. Assume that the first k operations performed by both algorithms are identical. Then, by Theorem 5.4, the answers to those operation will be the same. Since the $(k + 1)$-st operation depends only on the input and the results of the previous k operations, it must also be the same for both algorithms. Therefore the same sequence of data operations is performed in both algorithms, so $SP_2(d, t) = s$.

The proof that the second condition holds is the same as for Theorem 5.4. Either the input trail t contains the "correct" trail as a prefix, or one of the data structure operations will fail, resulting in an "error" output. ∎

One point has been glossed over in the above proof. In the SHORTEST-PATH algorithm results of deletemin(h) are not output nor are they stored in the certification trail. It might be possible for incorrect answers to be returned by deletemin(h) operations while still producing correct shortest paths and lengths. The second execution of the SHORTEST-PATH algorithm will not detect this since the correct output is produced. By proving that the answers to deletemin(h) operations are the same, we have proven more than strictly required.

6 Experimental Data on Certification Trails

We have performed extensive timing experiments on several basic and well-known problems, including the ones described in this paper. Algorithms for solving these problems were implemented, both

with and without the use of certification trails. Timing data was collected on both the certification trail solutions and the basic solutions. The following tables summarize these results.

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
5000	0.61	0.62	0.07	8.73	43.62
10000	1.33	1.34	0.14	9.56	44.54
25000	3.68	3.68	0.36	10.22	45.12
50000	7.68	7.74	0.71	10.75	44.94
100000	16.23	16.30	1.43	11.35	45.39
200000	33.93	34.37	2.84	11.94	45.16

Table 2: Convex Hull

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
10000	0.28	0.30	0.04	7.00	39.29
50000	1.80	1.90	0.19	9.47	41.94
100000	3.96	4.08	0.41	9.66	43.31
500000	23.95	24.69	2.14	11.19	43.99
1000000	50.23	51.57	4.38	11.47	44.31

Table 3: Sort

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
100,1000	0.04	0.05	0.02	2.00	12.50
250,2500	0.15	0.16	0.06	2.50	26.67
500,5000	0.31	0.33	0.11	2.82	29.03
1000,10000	0.70	0.76	0.23	3.04	29.29
2000,20000	1.58	1.67	0.45	3.51	32.91
2500,25000	2.06	2.15	0.55	3.75	34.47

Table 4: Shortest Path

The timing information was gathered on Sun SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during timing experiments.

Much of the data presented in the timing table is essentially self-explanatory relative to the certification trail technique and algorithms considered. However, a brief discussion of the table entries is appropriate.

The column labelled *Basic Algorithm* contains timing data which gives the execution time of the algorithm in producing the output without the generation of the certification trail. All timing data is listed in seconds.

The *First Execution* column gives the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Second Execution* column gives the execution time of the algorithm in producing the output while using the certification trail.

The *Speedup* column is the ratio of the run times of the Basic Algorithm and the Secondary Execution. One reason this figure is important is that it is possible for the two algorithms to run in different environments (different hardware, programming language, etc). A high speedup indicates that less powerful hardware or a higher level language (with associated overhead) may be sufficient for the second execution.

The *Percent Savings* column records the percentage of the execution time savings which is gained by using the certification trail method as compared to 2-version programming approach. The time required for a 2-version programming approach was estimated by doubling the time reported in the Basic algorithm. This assumes that both versions take approximately the same amount of time to execute.

In addition to the tables, the timing information for the convex hull algorithm is plotted in Figure 5. Plots for the other two examples are similar.

Examination of the data collected for the convex hull algorithm indicates that:

- The overhead in generating the certification trail is very small, less than 2% of the running time of the basic (no certification trail) algorithm.
- The second execution is very fast, achieving an order of magnitude speedup for larger input sizes. This suggests that a single "second algorithm" process could easily handle the output generated by several "first algorithm" processes running in parallel. Alternately, the high speedup would allow the second execution to be run on lower performance (and hence less expensive) hardward. Finally, the large speedup and reduced code complexity may make it possible to take advantage of a formally verifiable language (which may require significant overhead) in implementing the second algorithm.

The data for sorting indicates that the certification trail also requires very low overhead and results in a large speedup. For the shortest path problem the overhead is still very low, and the speedup, while not as dramatic as for the first two problems, is still quite respectable.

7 Comparison With Other Techniques

The certification trail approach shares similarities with other valuable fault tolerance and fault detection techniques that have been previously proposed and examined, but in each case there are significant and fundamental distinctions. These distinctions are primarily related to the generation and character of the certification trail and the manner in which the secondary algorithm uses the certification trail.

First consider the important and useful technique called N-version programming [9, 3]. When using this technique N different implementations of an algorithm are independently executed with subsequent comparison of the resulting N outputs. There is no relationship among the executions of the different versions of the algorithms other than that they all use the same input; each algorithm is executed independently without any information about the execution of the other algorithms. In marked contrast, the certification trail approach allows the primary algorithm to generate a trail of information which can be read by the secondary algorithm. The advantages of utilizing this additional information are shown in the body of this paper. In effect, N-version programming can be thought of relative to the certification trail approach as the employment of a *null trail.*

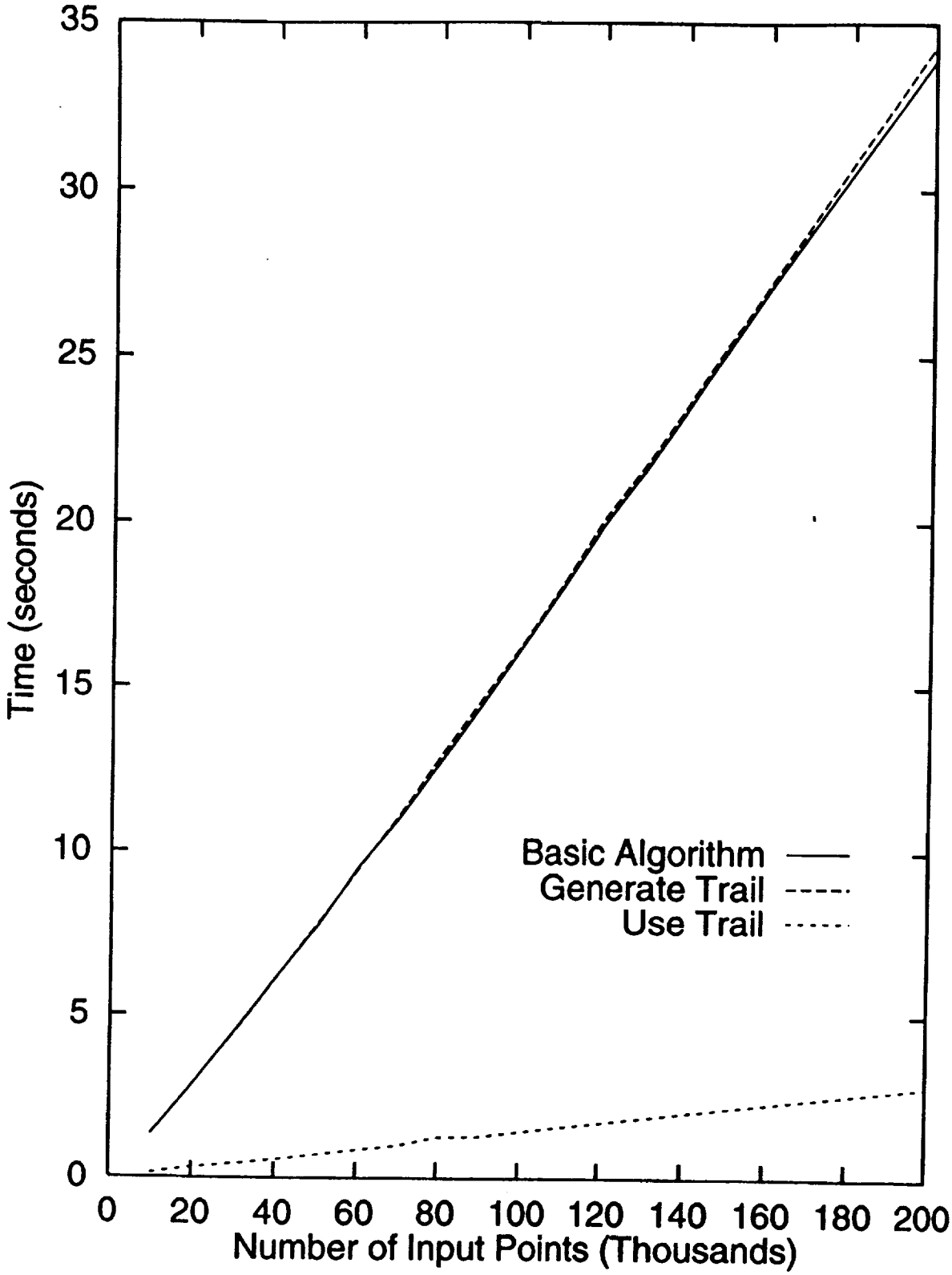


Figure 5: Convex Hull Run Times.

Another valuable technique, known as the recovery block approach [2, 18, 21], was proposed by Randell. It uses acceptance tests and alternative procedures to produce what is to be regarded as a correct output from a program. When using recovery blocks, a program is viewed as a being structured into blocks of operations, which after execution yield outputs which can be tested in some informal sense for correctness. The rigor, completeness, and nature of the acceptance test is left to the program designer, and many of the acceptance tests that have been proposed for use tend to be somewhat straightforward [2]. When using certification trails it is clearly possible to combine the second execution and the comparison test to yield a program which certifies the correctness of the output of the first execution. Unlike an acceptance test this certifier must satisfy strict formal properties of correctness. Also note that the certification trail technique emphasizes the capability of generating additional data to ease the certifying process and does not rely solely on data which would normally be computed. It should be possible to fruitfully combine the ideas of recovery blocks and certification trails.

Algorithm-based fault tolerance [15, 17, 19] uses error detecting and correcting codes for performing reliable computations with specific algorithms. This technique encodes data at a high level and algorithms are specifically designed or modified to operate on encoded data and produce encoded output data. Algorithm-based fault tolerance is distinguished from other fault tolerance techniques by three characteristics: the encoding of the data used by the algorithm; the modification of the algorithm to operate on the encoded data; and the distribution of the computation steps in the algorithm among computational units. The error detection capabilities of the algorithm-based fault tolerance approach are directly related to that of the error correction encoding utilized. The certification trail approach does not require that the data to be executed be modified nor that the fundamental operations of the algorithm be changed to account for these modifications. Instead, only a trail indicative of aspects of the algorithm's operations must be generated by the algorithm. As seen in Section 6, the production of this trail does not add significant overhead. Moreover, any combination of computational errors can be handled.

Recently, Blum and Kannan [6] have defined what they call a *program checker*. This interesting work has been followed by a burst of activity in this general area [12, 7, 25, 8, 4]. Each of these papers, however, describes work which differs significantly from the work we present. A program checker is an algorithm which checks the output of another algorithm for correctness. An early example of a program checker is the algorithm developed by Tarjan [23] which takes as input a graph and a supposed minimum spanning tree and indicates whether or not the tree actually is a minimum spanning tree.

The Blum-Kannan program checking method differs from the certification trail method in two important ways. First, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input/output specification of a problem and no assumptions are made about the method being used to solve the problem. Because of this the algorithm which is being checked is treated as a black box. It can not be altered nor can its internal status be examined and exploited. In the certification trail approach the algorithm being checked is not treated as a black box. Instead, the algorithm can be modified to generate additional information (i.e., the certification trail) which is considered to be useful in the checking/verification process. By exploiting this capability it is sometimes possible to design certification trail solutions which allow faster checking than Blum-Kannan program checkers. Of course, these faster solutions are more specialized than the Blum-Kannan checkers which are guaranteed to work for any algorithm which solves the original problem. We believe that the added speed often outweighs the disadvantage of specialization.

The second important difference concerns the number of times that the program which is being checked is executed. In the Blum-Kannan approach the program may be invoked a polynomial

number of times. In the certification trail approach the program is run only once. Thus, the overall time complexity of the checking process can be significantly larger for Blum-Kannan checkers.

A third less important difference stems from the fact that Blum-Kannan checkers are defined in a more general probabilistic context. Certification trails are currently defined only for deterministic programs and checkers. However, it is clearly possible to define them in the more general probabilistic context.

Other work has been done to extend the ideas of Blum-Kannan to give methods which allow the conversion of some programs into new programs which are self-testing and self-correcting [12, 7]. However, these methods are also based on treating programs as black boxes and thus have limitations similar to Blum-Kannan program checkers. A recent paper by Blum et al. [8] concerns checking the correctness of memories and data structures. The results described in that paper differ from our work using abstract data types in one central way. The checkers that they design are tightly constrained in memory usage. Typically, they use only $O(\log(n))$ storage to check data structures of size $O(n)$. Our results do not place space constraints on the algorithm used to certify the data structure. Without a space constraint we are able to certify abstract data types such as priority queues which are more complex than the data structures that they check, i.e., stacks and queues. Also, we are able to achieve a speed up in the checking process and they are not.

Babai, Fortnow, Levin and Szegedy [4] present methods which appear to allow remarkably fast checking, i.e., in polylogarithmic time. Their approach has some similarities to the methods we propose. Both methods modify original algorithms to yield new algorithms which output additional information. We refer to this additional information as a certification trail and they refer to this information as a *witness*. In our case we are interested in modified algorithms which have the same asymptotic time complexity as the original algorithm. Indeed, the modified algorithm should be slowed down by at most a factor of two. In [4] the modified algorithm is slowed down by more than any fixed multiplicative factor. Specifically, if the original algorithm has a time complexity of $O(T)$ then the modified algorithm has a time complexity of $O(T^{1+\epsilon})$. Note that in practice the ϵ cannot be too small because its inverse appears in the exponent of the checker time complexity. Another difference between our methods is the fact that their method requires that the input and output be encoded using an error-correcting code. The encoding process takes $O(N^{1+\epsilon})$ time for strings of length N. However, many of the checkers we have developed take only linear time so the cost of simply preparing to use their method appears to be too great in some cases. It is also necessary to decode the output after the check. Lastly, we note that Fortnow has stated that their result is currently not practical [24].

8 Generalization and Future Research Areas

The experimental timing data on certification trails indicates that this technique is of great practical value as well as of theoretical interest. Furthermore, the techniques illustrated are applicable to a wide range of problems, especially the certification of Abstract Data Types described in the shortest path example. There are many areas of interest for future exploration, a few of which are described below.

8.1 Certified Data Structure Libraries

It is apparent that the certification trail technique described for the SHORTEST-PATH program may be used for a variety of problems. Since the certification trail is produced and used by abstract data type operations, the technique may be used with any algorithm that can be implemented in terms of those abstract data types. Creating a library of such "certified data types" enables

programmers to create fault tolerant programs without having to be familiar with the certification trail technique. Object oriented programming appears to be well suited to this task.

A possible objection to this is that it provides fault detection only for the data structure implementation, since the surrounding code is simply reused. Furthermore, the data structure implementation is likely to come from library code, and hence be highly reliable. In answer to this note that:

- In many algorithms, the code using the data structure is much simpler than the code implementing the data structure.
- Although the example above illustrated reuse of using the data structures, it is certainly possible for this code to be developed separately for the first and second execution programs.
- Errors are often found even in code that has been in use for a long period of time. The added confidence of using this technique may be desirable even for library code.
- Even if the library code is highly reliable, the certification trail can be helpful in detecting errors caused by hardware problems.
- Library code may have to be tuned or even rewritten to meet for a particular application or environment, partially negating the claim of using well-tested code.

Even if fault detection is not an issue, the certification trail technique is useful during program testing and debugging. Input may be automatically generated and processed. If the output of the first and second executions differ or an error is otherwise flagged, the input set is flagged. This reduces the need to otherwise compute output for selected input and enables both more and larger sets of input to be processed. 2-version programming may be used during debugging in a similar manner, however certification trails have the advantage of reduced overhead, allowing more test cases to be run, a reduction in the hardware required for testing, or both.

8.2 Almost-concurrent execution of the certification trail

In the above discussion and examples, the certification trail programs have been executed serially, i.e., we do not run the second execution until after first execution completed. Actually, except for sorting, the two executions in the examples above can be run almost-concurrently. The "second" execution simply reads the information from the certification trail as it becomes available. The two programs will finish nearly simultaneously, the difference being in the time after the last element is read from or written to the certification trail.

8.3 Continuing after an error

A possible extension to the use of certification trails is to attempt to continue the second execution after an error is detected. Consider the shortest path example using abstract data types. In that example, the second execution used an indexed linked list that performed each operation in constant time by using the certification trail from the first execution. Suppose that an error had been detected during the second execution. Rather than simply aborting, it may be possible to continue execution. This could be done by

- Reorganizing the existing set into some other data structure (such an AVL tree, red-black tree, etc.) that allows efficient operation without a certification trail.

- Continuing to use the indexed linked list and ignoring the rest of the certification trail. Note that this would result in some operations requiring more time.
- Continuing to use the indexed linked list and attempting to use the certification trail for future operations. This may be possible if the error that occurred has sufficiently "local" effect. For example, if part of a tree structure is corrupted during the first execution, it is still possible that operations involving other parts of the tree will be performed correctly.

On a related topic, research has been done on "self-correcting" data structures in which enough redundancy is built into a data structure so that it may be reconstructed if part of it is corrupted. Using certification trails with such structures could provide an efficient detector for corruption of the data structure.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[4] Babai, L., Fortnow, L., Levin, L., and Szegedy, M., "Checking computations in polylogarithmic time, " *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 21-31, 1991.

[5] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[6] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[7] Blum, M., Luby, M., and Rubinfeld, R., "Self-testing/correcting with applications to numerical problems," *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pp. 73-83, 1990.

[8] Blum, M., Evans, W., Gemmell P., Kannan, S., and Naor, M., "Checking the correctness of memories," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science* pp. 90-99, 1991

[9] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[10] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[11] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, Sept., 1959.

[12] Gemmell, R., Lipton, R., Rubinfeld, R., Sudan, M., and Wigderson, A., "Self-testing/correcting for polynomials and for approximate functions," *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 32-42, 1991.

[13] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set", *Information Processing Letters*, pp. 132-133, 1, 1972.

[14] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[15] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

[16] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[17] Jou, J.-Y. and Abraham, J. "Fault tolerant FFT networks," *Dig. of the 1985 Fault Tolerant Computing Symposium*, pp. 338-343, IEEE Computer Society Press, June, 1985.

[18] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[19] Nair, V., and Abraham, J., "General linear codes for fault-tolerant matrix operations on processor arrays," *Dig. of the 1988 Fault Tolerant Computing Symposium*, pp. 180-185, June, 1988.

[20] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

[21] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[22] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[23] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[24] Paul Wallich, "Crunching Epsilon," *Scientific American*, pp. 22-24, Jan., 1993

[25] Andrew Chi-Chih Yao, "Coherent Functions and Program Checkers," *Proc. 22 ACM Symp. of Theory of Computing*, pp. 84-94.

Experimental Evaluation of the Certification-Trail Method

Gregory F. Sullivan,[1] Dwight S. Wilson,[2] Gerald M. Masson,[3]
Mamoru Itoh,[4] Warren W. Smith, Jonathan S. Kay[5]

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [1, 2, 3, 4]. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the method. The method is applied to algorithms for the following problems: huffman tree, shortest path, minimum spanning tree, sorting, and convex hull. Our results reveal many cases in which an approach using certification-trails allows for significantly faster overall program execution time than a basic time redundancy-approach.

We also examine algorithms for the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. We implemented and analyzed answer-validation solutions for two types of priority queues. In both cases, the algorithm which performs answer-validation is substantially faster than the original algorithm for computing the answers.

Next we present a probabilistic model and analysis which enables comparison between the certification-trail method and the time-redundancy approach. The analysis reveals some substantial and sometimes surprising advantages for the certification-trail method.

[1]Research partially supported by NSF Grants CCR-8910569 and CCR-8908092 and an IBM Technology Interchange Program Grant.

[2]Research partially supported by NSF Grant CCR-8910569 and an IBM Technology Interchange Program Grant.

[3]Research partially supported by NASA Grant NSG 1442 and an IBM Technology Interchange Program Grant.

[4]Visiting Scholar, Matsushita Electronic Components Co.

[5]Currently at Dept. of Computer Science, University California San Diego

Finally we discuss the work our group has performed on the design and implementation of fault injection testbeds for experimental analysis of the certification trail technique This work employs two distinct methodologies: software fault injection (modification of instruction, data, and stack segments of programs on a Sun Sparcstation ELC and on an IBM 386 PC) and hardware fault injection (control, address, and data lines of an Motorola MC68000-based target system pulsed at logical zero/one values). Our results indicate the viability of the certification trail technique. We also believe the tools we have developed provide a solid base for additional exploration.
Keywords: Software fault tolerance, certification trails, error monitoring, design diversity, data structures.

1 Introduction

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [1, 3]. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the method. We have implemented several fundamental algorithms together with versions of the algorithms which generate and utilize certification trails. Specifically, algorithms for the following problems are analyzed: huffman tree, shortest path, minimum spanning tree, sorting, and convex hull. Our results reveal many cases in which an approach using certification trails allows for significantly faster overall program execution time than a basic time redundancy approach.

We also examine algorithms for the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. For this paper we implemented and analyzed answer-validation solutions for two abstract data types. The first solution is for a simplified priority queue which allows insert, min and deletemin operations, and the second solution is for a priority queue which allows insert, min, delete and deletemin operations. In both cases, the algorithm which performs answer-validation is substantial faster than the original algorithm for computing the answers.

This paper next presents a simple probabilistic model and analysis which enables comparison between the certification-trail method and the time-

redundancy approach. The analysis shows that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the method has both a a smaller probability of error and a smaller probability of undetected error. Surprisingly, the analysis also reveals the intriguing result that the certification-trail method often can display superior performance even when the method has the same execution time or a longer execution time than the time-redundancy approach. This superior behavior stems from the typical assymetry of the execution times of the first and second executions in the certification-trail method.

The paper next discusses the work our group has performed on the design and implementation of fault injection testbeds. This work employs two distinct methodologies: software fault injection and hardware fault injection. The software fault injection tool is similar to an interactive debugger but more accurately can be considered an interactive bugger. It allows programs to be halted and faults to be injected by direct modification of the stack, data and instruction segments of a program. Output can then be captured and characterized.

The hardware fault injector is based on injecting faults into an operating microprocessor. The injection is performed by explicitly setting one or more pins of the microprocessor to logical zero and/or logical one values. The timing and duration of the pin setting is under control of a supervisory processor. The testbed also includes a multi-processor system. This system consists of three processors which are connected to one another pairwise by shared banks of dual ported memory. We plan to use this system to conduct evaluation of systems which utilize concurrent execution of algorithms using the certification-trail method.

2 Introduction to Certification Trails

To explain the essence of the certification-trail technique for software fault tolerance, we will first discuss a simpler fault-tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so-called time redundancy

[32, 52]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct. The second possibility, of undetected faults, occurs when the output of the execution is unaffected by the faults.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming [16, 12] (in this case N=2), allows for the detection of errors caused by some faults in the software in addition to those cause by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

The certification-trail technique is designed to obtain similar types of error-detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail.* This data is chosen so that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected in the data trail.

3 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

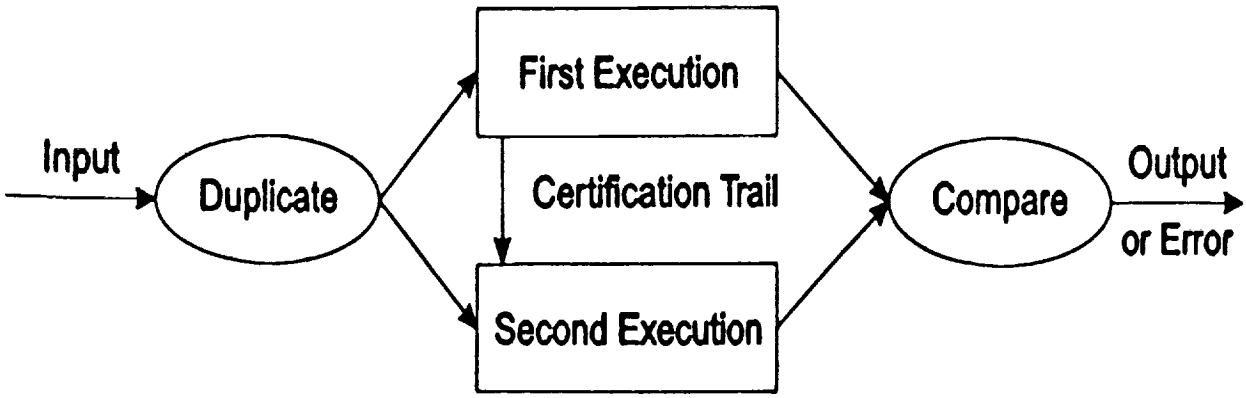


Figure 1: Certification trail method.

Definition 3.1 A problem **P** is formalized as a relation, i.e., a set of ordered pairs. Let **D** be the domain (that is, the set of inputs) of the relation **P** and let **S** be the range (that is, the set of solutions) for the problem. We say an algorithm **A** solves a problem **P** iff for all $d \in \mathbf{D}$ when d is input to **A** then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 3.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. **T** is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that
$F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$
either ($F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$) or $F_2(d, t) = \text{error}$.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. The definitions above assure that the error-detection capability of the certification-trail approach is similar to that obtained with the simple time-redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

Throughout this section we have assumed that our method is implemented with software, however, it is clearly possible to implement the method with assistance from dedicated hardware. The degree of diversity or independence achieved when using certification trails depends on how they are used. A fuller discussion of this and of the relationship between certification trails and other approaches to software fault tolerance is contained in the expanded version of [1].

4 Generalized Priority Queue

Before we present our example algorithms which use certification trails we must discuss the notion of an abstract data type. An abstract data type has a well defined data object or set of data objects, and an abstract data type has a carefully defined finite collection of operations that can be performed on its data object(s). Each operation takes a finite number of arguments (possibly zero), and some but not all operations return answers.

Some of the algorithms presented in the next section use the priority queue abstract data type. In addition, later in this paper the answer-validation problem for two variants of the priority queue are presented. Therefore, we now describe the priority queue. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, k is a key value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i,k) < (i',k')$ iff $k < k'$ or ($k = k'$ and $i < i'$). The abstract data types we will consider support a subset of the following operations.

member(i) returns a boolean value of true if the set contains an ordered pair with item number i, otherwise returns false.

insert(i,k) adds the ordered pair (i,k) to the set. We require that no other pair with item number i be in the set.

delete(i) deletes the unique ordered pair with item number i from the set. We require that a pair with item number i be in the set initially.

changekey(i, k) is executed only when there is an ordered pair with item number i in the set. This pair is replaced by (i, k).

deletemin (or deletemax) returns the ordered pair which is smallest (or largest) according to the total order defined above and deletes this pair. If the set is empty then the token "empty" is returned.

min (or max) returns the ordered pair which is smallest (or largest) according to the total order defined above. If the set is empty then the token "empty" is returned.

predecessor(i) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned. We require that a pair with item number i be in the set initially.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

Many different types and combinations of data structures can be used to support different subsets of these operations efficiently.

5 Examples of the Certification Trail Technique with Timing Data

In this section we evaluate the use of certification trails for five well-known and significant problems in computer science: the convex hull problem, the minimum spanning tree problem, the shortest path problem, the Huffman tree problem, and the sorting problem. We have implemented algorithms for these problems together with other algorithms which generate and use certification trails.

We provide a full description of the algorithm for the convex hull problem which generates a certification trail and a full description of the algorithm which uses that trail. This material has not appeared in our previous publications [1, 3]. Because of space considerations the discussion of three of the other algorithms is abbreviated, but references to previous publications or technical reports which describe the algorithms more fully are given. The treatment of the sort algorithm is brief but is detailed enough for the interested reader to implement the certification-trail method.

The algorithms we have choosen to implement are not always the algorithms which have the smallest asymptotic time complexity. Often the asymptotically fastest algorithms have large constants of proportionality which make them slower on the data sizes we examined. We modified and used some programs from major software distributions such as quicker-sort from a Berkeley Unix distribution. Other algorithms were based on textbook discussions. It should be stressed here that this research is exploratory and we hope to further increase our corpus of algorithm and data-structure implementations.

5.1 Systems used for timing data

We have collected timing data for the algorithms considered using a Sun workstation, an IBM 386 PC and a Motorola 68000-based system.

The SUN machine utilized was a SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during the timing experiments. Timing data was obtained through the getrusage() system call; the user times are reported in the data.

Some of the algorithms were also run on an MSDOS machine: a Northgate 386/33 with 8MB of RAM. The programs were compiled using DJGPP, DJ Delorie's port of the GNU GCC compiler to MSDOS. This compiler uses a DOS extender to allow programs to run in protected mode; thus nearly all of the 8MB in the machine was available, thereby allowing data sets comparable in size to those used on the Sun. The programs required no change to run under MSDOS, though the data generators required minor modification because the drand48() family of random number generators was not available.

Finally some of the algorithms were also run Motorola M68000-based target system. In addition to the MC68000 microprocessor which served as the cpu, the system was also was comprised of 512K bytes of RAM, 512 bytes of ROM, and numerous I/O modules to support serial and parallel communication. A timer module is also included in the system which uses the 4Mhz clock as a reference so as to provide execution time data for experiments. This system is discussed in Section 10 relative to fault injection experiments.

5.2 Explanation of timing data table entries

Much of the data presented in the timing table is essentially self-explanatory relative to the certication trail technique and algorithms considered. However, a brief discussion of the table entries is appropriate.

The *Basic Algorithm* timing data refers to the execution time of the algorithm in producing the output without the generation of the certification trail. All timing data is listed in seconds.

The *Generate Certif.* timing data refers to the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Use Certif.* timing data refers to the execution time of the algorithm in producing the output while using the certification trail.

The *Compare* timing data refers to the time necessary to compare the outputs from both two Basic Algorithm runs or from a Generate Certification Trial run and a Use Certification Trail run. (Obviously, the value of the comparison would be the same in each case.) For the some of the experiments, the data was too small to calculate and is therefore listed as 0.00. In other experiments, the comparison was included in the algorithm execution timing data and therefore is not separately listed.

The *Total Basic* timing data is twice the Basic Algorithm timing data plus the Comparison time (when available) so as to evaluate the classical time-redundancy approach.

The *Total Certif.* timing data is the sum of the Generate Certif. timing data and the Use Certif. data and Comparison data (when available) so as to evaluate the certification trail approach.

The *% Savings* data is percentage of the execution time savings which is gained by using the certification trail method as compared to the classical time redundancy method.

For the Huffman tree data, the input size for the Huffman tree program is the number of nodes. Each node is given a frequency, chosen uniformly from the integers $\{1, 2, \ldots, n\}$. n was selected to be the number of nodes, but in fact it's value does not affect the running time of the algorithm. In order for the algorithm to execute correctly, the sum of the frequencies must not cause an arithmetic overflow. The certification trail method will detect this.

For the minimum spanning tree and shortest path tables, there are two numbers associated with the input size, the first is the number of vertices in the graph, the second the number of edges. A graph with the required

edges is selected uniformly from the set of all such graphs, then tested for connectedness. The algorithms will function regardless of connectedness, but allowing graphs that are not connected would introduce undesirable variation in the timing data.

For the convex hull tables, the input size is the number of points in the data set. The points are chosen uniformly from the set of points with integer coordinates between 0 and 30,000.

For the sorting tables, sorting was timed in two ways. The first set of results were obtained by sorting integers. To generate a trail, an integer tag is added to each input integer and an array of these pairs passed to the sort function. After sorting, the "data" integers are placed in an array, and the "tag" integers are placed on the certification trail. Thus, the sort call looks the same as a normal sort function. The time to massage the data in this manner is included in the cost of the call. This method resulted in only a small speedup, because of the overhead involved in massaging the data, and because the sort routine must swap pairs of integers instead of single integers. The integers were chosen uniformly over the range 0 to 1,000,000.

The second method was to sort an array of pointers to structures. In this case it was assumed that the structure contained a field that would serve as the tag. The sort program needed only to fill in this field, and not copy the structures to a second array. This method results in dramatic speedups. Integer keys were used, though a more complex key will work as well (in fact, a more complex key is very likely to increase the speedup achieved).

For the priority queue and generalized priority queue tables, the input size n is the number of commands executed. The item numbers range from 1 to n (ie. there are as many item numbers as there are commands). The commands are not chosen with equal probability, but rather the first $n/2$ are weighted toward insert operations while the second half are weighted toward the other operations, the weightings remaining the same for all runs. This weighting is necessary in order to force a large queue.

The timing data displayed in the tables should be considered not only relative to the overall efficiencies of the certification trail method relative to classical time redundancy but also relative to the probabilistic analysis given in Section 9 in which we show that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the certification trail method has both a a smaller probability of error and a smaller probability of undetected error.

5.3 Convex Hull Example

The convex hull problem is a fundamental one in computational geometry. Our certification trail solution is based on a solution due to Graham [24] which is called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos[46]. For simplicity in the discussion which follows we will assume the points are in so called general position, e.g., no three points are colinear. It is not hard to remove this restriction.

Definition 5.1 The *convex hull* of a set of points, S, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique and its vertices are a subset of the points in S. It is specified by a counterclockwise sequence of its vertices.

Figure 2(c) shows a convex hull for the points indicated by black dots. The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects an "extreme" point and calls it p_1. The next two steps sort the remaining points in a way which is depicted in Figure 2(a). It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although this polygon may not be convex. It is possible to think of the algorithm as removing points from this simple polygon until it becomes convex. The main FOR loop iteration adds vertices to the polygon under construction and the inner WHILE loop removes vertices from the construction. A point is removed when the angle test performed at line 6 reveals that it is not on the convex hull because it falls within the triangle defined by three other points. A "snapshot" of the algorithm given in Figure 2(b) shows that q_5 is removed from the hull. The angle formed by q_4, q_5, p_6 is less than 180 degrees. This means, q_5 lies within the triangle formed by q_4, p_1, p_6. (Note, $q_1 = p_1$.) In general, when the angle test is performed if the angle formed by q_{m-1}, q_m, p_k is less than 180 degrees then q_m lies within the triangle formed by q_{m-1}, p_1, p_k. Below it will be revealed that this is the main fact that our certification trail relies on. When the main FOR loop is complete the convex hull has been constructed.

Algorithm CONVEXHULL(S)
Input: Set of points, S, in R^2

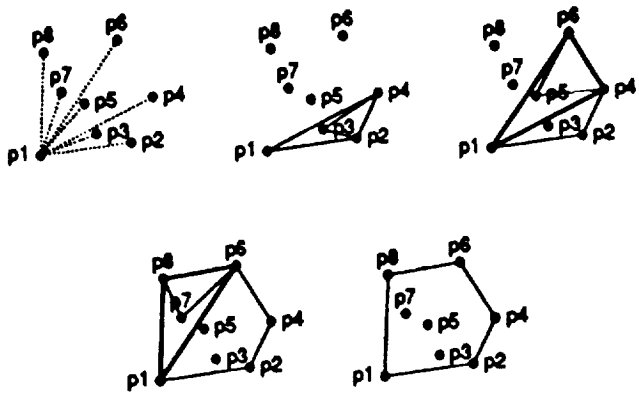


Figure 2: Convex hull example.

Output: Counterclockwise sequence of points in R^2 which define convex hull of S
1 Let p_1 be the point with the largest x coordinate (and smallest y to break ties)
2 For each point p (except p_1) calculate the slope of the line through p_1 and p
3 Sort the points (except p_1) from smallest slope to largest. Call them $p_2, \ldots, p_n$
4 $q_1 := p_1$; $q_2 := p_2$; $q_3 := p_3$; $m = 3$
5 FOR $k = 4$ to n DO
6 WHILE the angle formed by q_{m-1}, q_m, p_k is ≥ 180 degrees DO $m := m - 1$ END
7 $m := m + 1$
8 $q_m := p_k$
9 END FOR
10FOR $i = 1$ to m DO, OUTPUT(q_i) END FOR
END CONVEXHULL

First execution: In this execution the code CONVEXHULL is used. The certification trial is generated by adding an output statement within the WHILE loop. Specifically, if an angle of less than 180 degrees is found in the WHILE loop test then the four tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The table below shows the four tuples of points that would be output by the algorithm when run on the example in Figure 2. The points in the table are given the same names as in Figure 2(a). The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Strictly speaking the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $s_1, s_2, \ldots, s_n$ then rather than ouput the element in R^2 corresponding to s_i the number i is output. It is not hard to code the program so that this is done.

Point not on convex hull	Three surrounding points
p_5	p_4, p_1, p_6
p_4	p_3, p_1, p_6
p_7	p_6, p_1, p_8

Second execution: Let the certification trail consist of a set of four tuples, $(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r)$ followed by the supposed convex hull, $q_1, q_2, \ldots, q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

- First, the algorithm checks for $i \in \{1, \ldots, r\}$ that x_i lies within the triangle defined by a_i,b_i, and c_i.
- Second, the algorithm checks that for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is less than or equal to 180 degrees.
- Third, it checks that there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.
- Fourth, it checks that for $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.
- Fifth, it checks that there is a unique point among the points on the supposed convex hull which is a local extreme point. We say a point q on the hull is a *local extreme* point if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; instead it makes them easier. The correctness and adequacy of these checks must be proven. Because of space limitations we shall not give the proof here.

Time complexity: In the first execution the sorting of the input points takes $O(n \log(n))$ time where n is the number of input points. One can show that this cost dominates and the overall complexity is $O(n \log(n))$.

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.74	0.79	0.11	0.03	1.51	0.93	38.41
20000	1.65	1.75	0.23	0.06	3.36	2.05	39.28
50000	4.64	4.79	0.59	0.14	9.42	5.52	41.40
100000	9.95	10.32	1.19	0.28	20.18	11.79	41.57

Table 1: Huffman Tree on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.09	1.32	0.32	0.10	2.28	1.74	23.68
20000	2.38	2.91	0.63	0.21	4.97	3.75	24.55
50000	7.01	8.80	1.59	0.50	14.52	10.89	25.00

Table 2: Huffman tree on 386/33

It is possible to implement the second execution so that all five checks are done in $O(n)$ time. /papers/certify3/tabdata /papers/certify3/tabdataChecking that a point lies within a triangle is a geometric calculation that can be done in constant time. Comparing the angle formed by three points to 180 degrees can be done in constant time. The third and fourth checks can be done in $O(n)$ because the certification trail contains indices into the input data as described above. The uniqueness of the "local extreme" can also be checked in linear time.

5.4 Minimum Spanning Tree Example

This classic problem has been examined extensively in the literature and an historical survey is given in [25]. Our approach is applied to a variant

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.26	1.29	0.13	0.01	2.53	1.43	43.47
20000	2.71	2.81	0.31	0.01	5.43	3.13	42.35
50000	7.41	7.48	0.70	0.01	14.83	8.19	44.77
100000	15.76	15.87	1.43	0.01	31.53	17.31	45.09

Table 3: Convex Hull on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.79	1.88	0.15	0.01	3.59	2.04	43.18
20000	3.86	4.08	0.31	0.01	7.73	4.40	43.08
50000	10.51	11.16	0.78	0.01	21.03	11.95	43.18
100000	22.40	23.97	1.64	0.01	44.81	25.62	42.83

Table 4: Convex Hull on 386/33

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
100,1000	0.04	0.05	0.01	0.00	0.08	0.06	25.00
200,2000	0.10	0.12	0.02	0.00	0.20	0.14	30.00
500,5000	0.30	0.31	0.06	0.00	0.60	0.37	38.33
1000,10000	0.68	0.72	0.13	0.00	1.36	0.85	37.50
1500,15000	1.10	1.14	0.19	0.00	2.20	1.33	39.55
2000,20000	1.51	1.58	0.27	0.00	3.02	1.85	38.74
2500,25000	1.97	2.00	0.35	0.00	3.94	2.35	40.36

Table 5: Minimum Spanning Tree on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
100,1000	0.04	0.03	0.01	0.00	0.08	0.04	50.00
200,2000	0.08	0.08	0.02	0.00	0.16	0.10	37.50
500,5000	0.26	0.24	0.06	0.00	0.52	0.30	42.31
1000,10000	0.59	0.56	0.13	0.00	1.18	0.69	41.53
1500,15000	0.93	0.90	0.20	0.00	1.86	1.10	40.86
2000,20000	1.29	1.28	0.28	0.00	2.58	1.56	39.53
2500,25000	1.67	1.65	0.36	0.00	3.34	2.01	39.82

Table 6: Shortest Path on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.23	0.40	0.06	0.01	0.47	0.47	0.00
20000	0.51	0.86	0.13	0.01	1.02	1.00	1.96
50000	1.38	2.35	0.35	0.02	2.78	2.72	2.15
100000	2.96	4.97	0.76	0.04	5.92	5.73	3.20

Table 7: Integer sorting on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.02	1.18	0.14	0.04	2.08	1.36	34.62
20000	2.16	2.49	0.29	0.08	4.40	2.86	35.00
50000	5.67	6.48	0.73	0.22	11.56	7.43	35.73
100000	11.74	13.48	1.57	0.44	23.92	15.49	35.24

Table 8: Integer Sort on 386/33

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.32	0.33	0.03	0.01	0.65	0.37	43.07
20000	0.71	0.72	0.07	0.01	1.43	0.80	44.05
50000	1.97	1.99	0.18	0.02	3.96	2.19	44.69
100000	4.32	4.37	0.38	0.05	8.69	4.80	44.76

Table 9: Pointer sorting on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.08	1.15	0.07	0.03	2.19	1.25	42.92
20000	2.41	2.41	0.16	0.07	4.89	2.64	46.01
50000	6.37	6.38	0.42	0.22	12.96	7.02	45.83
100000	13.29	13.33	0.89	0.43	27.01	14.65	45.76

Table 10: Pointer Sort on 386/33

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.86	0.83	0.14	0.01	1.73	0.98	43.35
20000	1.92	1.87	0.28	0.01	3.85	2.16	43.89
50000	5.32	5.37	0.69	0.02	10.64	6.08	42.85

Table 11: Data structs on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Total Basic	Total Certif.	% Saving
8	0.075	0.091	0.026	0.151	0.117	28.7
16	0.215	0.248	0.054	0.430	0.302	42.4
32	0.561	0.629	0.111	1.122	0.740	51.6
64	1.330	1.468	0.224	2.660	1.692	57.2
128	3.120	3.398	0.450	6.240	3.848	62.2
256	7.225	7.783	0.903	14.450	8.686	66.4
512	16.270	17.388	1.808	32.540	19.196	69.5

Table 12: Huffman Tree on 68000-based system

Size		Basic	Generate	Use	Total	Total	% Saving
Nodes	Edges	Algorithm	Certif.	Certif.	Basic	Certif.	
10	15	0.053	0.054	0.055	0.106	0.109	-2.5
10	20	0.071	0.072	0.073	0.142	0.145	-1.7
10	25	0.088	0.089	0.090	0.176	0.179	-1.5
50	75	0.320	0.323	0.309	0.639	0.632	1.2
50	100	0.423	0.427	0.400	0.846	0.826	2.3
50	125	0.492	0.496	0.464	0.984	0.960	2.5
100	150	0.652	0.658	0.602	1.305	1.260	3.6
100	200	0.874	0.881	0.789	1.748	1.671	4.6
100	250	1.036	1.045	0.938	2.073	1.983	4.5
500	750	3.588	3.617	3.047	7.176	6.664	7.7
500	1000	4.780	4.817	3.955	9.560	8.772	9.0
500	1250	5.656	5.698	4.717	11.311	10.415	8.6
1000	1500	7.474	7.533	6.115	14.949	13.649	9.5
1000	2000	9.902	9.977	7.919	19.803	17.895	10.7
1000	2500	11.830	11.917	9.517	23.660	21.434	10.4
1500	2250	11.415	11.503	9.157	22.830	20.660	10.5
1500	3000	14.967	15.077	11.802	29.933	26.879	11.4

Table 13: Min Spanning Tree on 68000-based system

of the Prim/Dijkstra algorithm [47, 18] as explicated in [54]. We provide a definition of the problem below. For more information on the graph theoretic terminology used in this problem and others the reader may consult [54, 17].

Definition 5.2 Let $G = (V, E)$ be a graph and let w be a positive rational valued function defined on E. A *subtree* of G is a tree, $T(V', E')$, with $V' \subseteq V$ and $E' \subseteq E$. We say T *spans* V' and V' is spanned by T. If $V' = V$ then we say T is a spanning tree of G. The weight of this tree is $\sum_{e \in E'} w(e)$. A minimum spanning tree is a spanning tree of minimum weight.

The problem is to input a graph with edge weights and output a minimum spanning tree. The algorithm for this problem which has the fastest asymptotic time complexity uses fusion trees and is given in [20]. This algorithm however appears to have a large constant of proportionality. Other asymptotically fast algorithms [22] also appear to be handicapped by large constants of proportionality. A fuller discussion of the two algorithms we employ for generation and use of a certification trial is given in [1].

5.5 Shortest Path Example

This is another classic problem which has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [18] as explicated in [54]. We are concerned with the single source problem, i.e., given a graph and a vertex s, find the shortest path from s to v for every vertex v.

The algorithm for this problem which has the fastest asymptotic time complexity uses fusion trees and is given in the same paper which we cited earlier when considering the minimum spanning tree problem[20]. This algorithm however appears to have a large constant of proportionality. Our solution employing the certification trail method is very closely based on the solution we gave for the minimum spanning tree problem [1].

5.6 Huffman Tree Example

This is another old algorithmic problem and one of the original solutions was found by Huffman[30]. It has been used extensively to perform data compression through the design and use of so called Huffman codes. These codes are prefix codes which are based on the Huffman tree and which yield excellent data compression ratios. The tree structure and the code design are based on the frequencies of individual characters in the data to

be compressed. Here we are concerned exclusively with the Huffman tree. See [30] for information about the coding application.

Definition 5.3 The Huffman tree problem is the following: Given a sequence of frequencies (positive integers) $f[1], f[2], \ldots, f[n]$, construct a tree with n leaves and with one frequency value assigned to each leaf so that the weighted path length is minimized. Specifically, the tree should minimize the following sum: $\sum_{l_i \in \text{LEAF}} \text{len}(i) f[i]$ where LEAF is the set of leaves, $\text{len}(i)$ is the length of the path from the root of the tree to the leaf l_i, $f[i]$ is the frequency assigned to the leaf l_i.

The method we employ to generate and use a certification trail is detailed in the following technical report [2].

5.7 Sorting Example

This important problem has a massive literature. In this section we will discuss how to apply the certification trail approach to the sorting problem. Let us assume that the sorting algorithm takes as input an array of n elements and outputs an array of n elements. The algorithm is supposed to place the data into non-decreasing order.

To design a certification trail algorithm we must discover the nature of the data that should be included in the certification trail to allow quick computation of the final output sorted array. Suppose that we decide to use the output array itself as the certification trail. We note that it is easy to check that this array is in non-decreasing order by simply performing a single pass over the array. Unfortunately, it is considerably more difficult to make sure that this array contains exactly the same elements as the original input array. Indeed, this problem has a lower bound time complexity of $\Omega(n \log(n))$ in a comparison based model.

Because of this difficulty we use the permutation of the elements defined by the input and output data arrays as the certification trail. To compute this permutation we allocate a new array of size n called permute which is initialized by setting its ith element to i. (Alternatively, we add a new field to pre-existing structures when structures are being sorted.) Each time the sort algorithm exchanges two elements the corresponding elements in the permute array are also exchanged. (If structures are being used then this happens automatically.) This approach works with all sort algorithms which are based on exchanging array elements. The code below shows how

the permute array is used to rapidly recompute the final sorted output array and how the permute array itself is checked.

Algorithm SORT USING TRAIL
Input: Arrays indata[1..n] and permute[1..n]
Output: outdata[1..n] containing the data in indata sorted into non-decreasing order

The first part of the algorithm checks that the permute values are in the proper range and constructs the output array.

```
1  FOR i := 1 to n DO
2      IF permute[i] > n or permute[i] < 1
3          THEN OUTPUT("Error: not a permutation") STOP
4          ELSE outdata[i] := indata[permute[i]]
5  END FOR
```

The next part of the algorithm checks that the output array is properly ordered.

```
6  FOR i := 2 to n DO
7      IF outdata[i − 1] > outdata[i] THEN OUTPUT("Error: decreasing value") STOP
8  END FOR
```

The final part of the algorithm checks that the permute array defines a proper permutation, i.e., each element is mapped to exactly one element.

```
9  FOR i := 1 to n DO present[i] = FALSE END
10 FOR i := 1 to n DO
11     IF present[permute[i]] = TRUE
12         THEN OUTPUT("Error: not a permutation") STOP
13         ELSE present[permute[i]] := TRUE
14 END FOR
END SORT USING TRAIL
```

Our experimental work on the Sun was based on a variant of quicksort [26] which is called quickersort [50]. The implementation of this algorithm that we used was provided by a Berkeley UNIX software distribution for the Sun. Our experimental work on the IBM PC was based on a quicksort algorithm implemented as part of a Gnu library of functions.

6 Answer-Validation Problem for Abstract Data Types

The next few sections of this paper are concerned with the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. Because of space limitations we will not discuss the details of how this can be done.

Below, we define the answer-validation problem. Next, we give two example algorithms for the answer-validation problems. The first algorithm is for a priority queue which allows insert, min and deletemin operations. The second algorithm is for a priority queue which allows insert, min, delete and deletemin operations. In the next section experimental data on the execution times of these algorithms is presented.

For each abstract data type we define an *answer-validation* problem. Intuitively, the answer validation problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on the abstract data type. More formally, the input to the answer-validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. In addition, the sequence contains the supposed answers for each of the operations which return answers. In particular, each supposed answer is paired with the operation that is supposed to return it. Examples of such inputs are given in the columns labelled "Operation" and "Answer" table 15.

The output for the answer-validation problem is the word "correct" if the answers given in the input match the answers that would be generated by actually performing the operations. The output is the word "incorrect" if the answers do not match. It is also useful to allow the output word to say "ill-formed". This output is used if the sequence of operations is ill-formed, e.g., an operation has too many arguments or an argument refers to an inappropriate object.

The answer-validation problem is similar to the idea of an acceptance test which is used in the recovery-block approach [48, 6] to software fault tolerance. The main difference is that an answer-validation problem is dependent upon a sequence of answers, not just an individual answer. Hence, if an incorrect answer appears in the sequence, it may not be detected immediately. It is guaranteed, however, that an incorrect answer will be detected at some point during the processing of the entire sequence. By allowing

for this latency in detection, it is possible to create a much more efficient procedure for solving the answer-validation problem.

The most important aspect of the answer-validation problem is that it is often possible to check the correctness of the answers to a sequence of operations much more quickly than actually calculating what the answers should be from scratch. In other words, the answer-validation problem has a smaller time complexity than the original abstract-data-type problem. This speedup is very useful in fault-detection applications.

It is possible to run an answer-validation algorithm for some abstract data type concurrently with some algorithm which uses the abstract data type. The answer-validation algorithm could act as a monitor making sure that all interactions with the abstract data type are handled correctly. This is valuable because many algorithms spend a large fraction of their time operating on abstract data types. Note, the overhead of this monitor is less than the overhead of actually performing the data-type operations a second time.

7 Answer Validation for Priority Queue

We will first consider the priority-queue abstract data type which allows only three operations: insert, min and deletemin. An example of a sequence of such operations appears in table 14. Many different data structures can be used to implement priority queues including heaps [61]; and balanced search trees such as AVL trees [5], red-black trees [27], or b-trees [13]. It is possible to process a sequence of $O(n)$ operations in $O(n \log(n))$ time using the data structures above. Furthermore, there is a lower bound of $\Omega(n \log(n))$ because it is possible to sort using a priority queue. Remarkably, the answer-validation problem can be solved using only $O(n)$ time, as documented below.

The algorithm which we present in this section is the same as that given in [3]. It is necessary to include a description of this algorithm because the algorithm in the next section (which has not appeared before) builds on this algorithm.

Each operation is time-stamped, i.e., the operations are assigned integers sequentially starting with 1 which is easy to do with a counter. The answer-validation algorithm uses a stack called answerstack. The contents of this stack are illustrated in table 14. The top of the stack is on the left in table 14.

Let us consider the kinds of tests that an answer-validation algorithm

Time	Operation	Answer	Insert time	Stack used in validation
1	insert(6,300)			
2	insert(2,404)			
3	insert(3,250)			
4	deletemin	(3,250)	3	(3,250,4)
5	insert(10,248)			
6	insert(12,245)			
7	insert(4,260)			
8	min	(12,245)	6	(12,245,8), (3,250,4)
9	insert(13,140)			
10	insert(5,142)			
11	deletemin	(13,140)	9	(13,140,11), (12,245,8), (3,250,4)
12	deletemin	(5,142)	10	(5,142,12), (12,245,8), (3,250,4)
13	deletemin	(12,245)	6	(12,245,13),(3,250,4)
14	deletemin	(10,248)	5	(10,248,14),(3,250,4)
15	deletemin	(4,260)	7	(4,260,15)

Table 14: Sequence of Priority Queue operations illustrating answer validation algorithm

for a priority queue might perform. Suppose (i,k) is the answer to some min or deletemin operation. Further, suppose (i′,k′) was the answer to a previous min or deletemin operation. If the priority queue is correct then either (i,k)≥(i′,k′) or (i,k) was inserted after the answer (i′,k′) was given. ** multiple insertions possible?* This suggests that the time of insertion for an element and the time of an answer should be recorded and the algorithm below does this. Unfortunately, if an algorithm compares an ordered pair which has been given as an answer against all previous answers then the algorithm complexity is at least $O(m^2)$. To avoid this a stack called the answerstack is used. The answerstack was designed to allow many comparisons to be done implicitly and thus the overall complexity of the many tests is reduced.

Algorithm for Answer Validation for Priority Queue

Input: Sequence of m operations together with arguments and supposed answers for the priority-queue data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Array called *inserttime* indexed by item number. Array elements contain either "absent" or a time-stamp. Array called *keyvalue* indexed by item number. Array elements contain either "absent" or a key value. Initially, each element in these two arrays contains "absent". Stack of ordered triples called *answerstack*. Each ordered triple has the following form: first element is an item number, second element is a key value, and third element is a time-stamp. answerstack is initially empty.

First phase: In this phase we process each operation as it appears serially using the following rules:

Let currenttime refer to the time-stamp of the operation being processed.

insert(i,k): If inserttime[i]≠"absent" then output "ill-formed" and stop. Otherwise, let inserttime[i] = currenttime and let keyvalue[i]=k.

min (i,k): (where (i,k) is the supposed answer to the deletemin operation.) If inserttime[i]="absent" or keyvalue[i]≠k then output "ill-formed" and stop.

Otherwise, let (i′,k′) be the item number and key value of the triple on the top of answerstack (if there is one). Repeatedly pop the stack until (i,k)<(i′,k′) or until answerstack is empty.

If answerstack is empty then push the triple (i,k,currenttime) onto answerstack and process the next priority queue operation.

If answerstack is non-empty then let the top element be (i',k',answertime'). If inserttime[i]<answertime' then output "incorrect" and stop. Otherwise, push the triple (i,k,currenttime) onto answerstack and process the next priority queue operation.

deletemin (i,k): (where (i,k) is the supposed answer to the deletemin operation.) Perform the same actions as those described for the min operation. However, just before processing the next priority queue operation, let inserttime[i]="absent" and let keyvalue[i]="absent".

Second phase: In this phase we operate on the items which have been inserted but have never been deleted.

Scan the array inserttime and for each item number for which inserttime[i]≠"absent" construct an ordered triple (i,keyvalue[i],inserttime[i]). Call this set of ordered triples remainders.

Use a bucket sort to sort the triples in remainders by their time-stamps, i.e., the third element of the ordered triple.

Merge the triples in remainders together with the triples in answerstack so that they are all ordered by their time-stamps, i.e., the third element of the ordered triple.

Scan the combined triples to determine if there exist two triples which satisfy the following: inserttime[i]<answertime' and (i,keyvalue[i])<(i',k'); where one triple is from remainders and has the form (i,keyvalue[i],inserttime[i]) and where the other triple is from answerstack and has the form (i',k',answertime');

If these two triples exist then output "incorrect" and stop. Otherwise output "correct" and stop.

Theorem 7.1 *The algorithm for answer validation of the priority queue abstract data type is correct.*

Theorem 7.2 *The answer validation algorithm for priority queue has a time complexity of O(n) for processing a sequence of O(n) operations.*

For proofs of these theorems see [3].

8 Answer Validation for Generalized Priority Queue

We next consider the priority-queue abstract data type which allows four operations: insert, min, deletemin, and delete. An example of a sequence of such operations appears in table 15.

The algorithm to solve the validation problem for this data type is an enhanced version of the algorithm given above for the data type which allowed only three priority-queue operations.

Algorithm for Answer Validation for Generalized Priority Queue

Input: Sequence of m operations together with arguments and supposed answers for the priority-queue data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: All the declartions used in the earlier algorithm are used again. In addition, a collection of sets called *stacksets* are used. Each set in stacksets consists of a set of item numbers (possibly the empty set). There is a one-to-one correspondence between the sets in stacksets and the ordered triples in answerstack. Initially, answerstack consists solely of the ordered triple (0,-∞,-1). Also initially, stacksets contains exactly one set which is the empty set and which corresponds to (0,-∞,-1).

First phase: In this phase we process each operation as it appears serially using the following rules:

Let currenttime refer to the time-stamp of the operation being processed.

insert(i,k): Perform the same actions as those given earlier for the insert operation. In addition, add the item number i to the set in stacksets corresponding to the top element in answerstack.

min (i,k): (where (i,k) is the supposed answer to the deletemin operation.) Perform the same actions as those given earlier for the min operation. In addition, if any elements are popped off of answerstack then the sets in stacksets corresponding to these elements are unioned together to form a new set. This new set is placed in correspondence with the new top element of answerstack.

deletemin (i,k): (where (i,k) is the supposed answer to the deletemin operation.) Perform the same actions as those given for the min operation described immediately above. In addition, remove the item number i from the set in stacksets which contains it. Further, before processing

Time	Operation	Answer	Insert time	Stack used in validation		
1	insert(5,310)					(0,-∞,-1) {5}
2	insert(6,210)					(0,-∞,-1) {5,6}
3	insert(8,280)					(0,-∞,-1) {5,6,8}
4	min	(6,210)	2			(6,210,4) {5,6,8}
5	insert(9,190)					(6,210,4) {5,6,8,9}
6	min	(9,190)	5		(9,190,6),	(6,210,4) {5,6,8,9}
7	insert(2,275)				(9,190,6), {2},	(6,210,4) {5,6,8,9}
8	delete(8)		3		(9,190,6), {2},	(6,210,4) {5,6,9}
9	insert(12,170)				(9,190,6), {2,12},	(6,210,4) {5,6,9}
10	insert(14,400)				(9,190,6), {2,12,14},	(6,210,4) {5,6,9}
11	deletemin	(12,170)	9	(12,170,11),	(9,190,6), {2,14},	(6,210,4) {5,6,9}
12	insert(3,290)			(12,170,11), {3},	(9,190,6), {2,14},	(6,210,4) {5,6,9}
13	insert(7,330)			(12,170,11), {3,7},	(9,190,6), {2,14},	(6,210,4) {5,6,9}
14	insert(15,200)			(12,170,11), {3,7,15},	(9,190,6), {2,14},	(6,210,4) {5,6,9}
15	delete(9)		5	(12,170,11), {3,7,15},	(9,190,6), {2,14},	(6,210,4) {5,6}
16	deletemin	(15,200)	14		(15,200,16), {2,3,7,14},	(6,210,4) {5,6}
17	delete(7)		13		(15,200,16), {2,3,14},	(6,210,4) {5,6}
18	deletemin	(6,210)	2			(6,210,18) {2,3,5,14}
19	delete(14)		10			(6,210,18) {2,3,5}

Table 15: Sequence of Priority Queue operations illustrating answer validation algorithm

the next priority queue operation, let inserttime[i]="absent" and let keyvalue[i]="absent".

delete(i): If inserttime[i]="absent" or keyvalue[i]="absent" then output "ill-formed" and stop.

Otherwise, let inserttime=inserttime[i] and let k=keyvalue[i]. Next, let inserttime[i]="absent" and let keyvalue[i]="absent".

Now, let (i',k',answertime') be the ordered triple which corresponds to the set in stacksets containing item number i. Next, remove item number i from the set which contains it.

If answertime'>inserttime and (i,k)>(i',k') then output "incorrect" and stop.

If answertime'>inserttime and (i,k)$\leq$(i',k') then process the next priority queue operation.

If (i',k',answertime') is the top element of answerstack then process the next priority queue operation.

Let (i'',k'',answertime'') be the element immediately above (i',k',answertime') on answerstack.

If (i,k)>(i'',k'') then output "incorrect" and stop. Otherwise, process the next priority queue operation.

Second phase: In this phase we operate on the items which have been inserted but have never been deleted.

For this phase one performs the same operations as the second phase described earlier.

Theorem 8.1 *The algorithm above for answer validation of the priority queue abstract data type is correct.*

Theorem 8.2 *The answer validation algorithm above for priority queue has a time complexity of O(n) for processing a sequence of O(n) operations.*

Proofs omitted for space reasons. It is clear that a priority queue with operations insert, delete, max, deletemax can also be validated in linear time by changing the appropriate signs in the algorithm above.

Definition 8.3 Consider a sequence of priority queue operations together with arguments and supposed answers. The sequence may contain the following operations: insert, delete, min, deletemin, max, and deletemax.

Based on this sequence we define a new sequence called a *minimum sequence*. This sequence differs from the original sequence as follows: Each max operation and answer pair is removed from the sequence. Each deletemax operation and answer pair is replaced by a delete(i) operation where i is the item number given in the answer to the deletemax operation. Each other operation remains the same.

We also define a *maximum sequence*. This sequence differs from the original sequence as follows: Each min operation and answer pair is removed from the sequence. Each deletemin operation and answer pair is replaced by a delete(i) operation where i is the item number given in the answer to the deletemin operation. Each other operation remains the same.

Theorem 8.4 *Consider a sequence of priority queue operations together with arguments and supposed answers. The sequence may contain the following operations: insert, delete, min, deletemin, max, and deletemax. The answers given for this sequence are correct if and only if the answers given for the corresponding minimum and maximum sequences are both correct.*

This theorem allows us to define an algorithm which solves the answer-validation problem for general priority queue.

9 Probabilistic Model

We will now present a simple probabilistic model with accompanying analysis which will permit a comparison between of our certification-trail method and the classical time-redundancy approach [32, 52]. The analysis shows that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the certification trail method has both a a smaller probability of error and a smaller probability of undetected error. Surprisingly, the analysis also reveals the intriguing result that the certification-trail method often can display superior performance even when the method has the same execution time or a longer execution time than the time-redundancy approach. This superior behavior stems from the typical assymetry of the execution times of the first and second executions in the certification-trail method.

We make the following assumptions.

i. Errors are distributed exponentially with parameter λ.

ii. If errors occur during only one phase of the execution, then they are detected.

iii. If errors occur in both phases of an execution they are not detected.

For solutions to a problem with run times a and b, we therefore have:

$$\begin{aligned} Pr\{correct\} &= e^{-\lambda(a+b)} \\ Pr\{detected\} &= e^{-\lambda a}(1-e^{-\lambda b}) + e^{-\lambda b}(1-e^{-\lambda a}) \\ &= e^{-\lambda a} + e^{-\lambda b} - 2e^{-\lambda(a+b)} \\ Pr\{undetected\} &= (1-e^{-\lambda a})(1-e^{-\lambda b}) \\ &= 1 - e^{-\lambda a} - e^{-\lambda b} + e^{-\lambda(a+b)} \\ &= 1 - Pr\{correct\} - Pr\{detected\} \end{aligned}$$

Given two solutions for a problem, we say that the first is strictly superior to the second iff:

$$\begin{aligned} &Pr_1\{correct\} \geq Pr_2\{correct\} \quad and \quad Pr_1\{undetected\} < Pr_2\{undetected\} \\ &\qquad\qquad\qquad\qquad or \\ &Pr_1\{correct\} > Pr_2\{correct\} \quad and \quad Pr_1\{undetected\} \leq Pr_2\{undetected\} \end{aligned}$$

This implies that the run time of the first solution is no greater than that of the second solution.

Observation 1 *Suppose there are two solutions (using certification trails) to a problem, such that each solution runs in two phases, and the combined run times of phases is the same for both solutions. Then the solution with the greater time imbalance between phases is strictly superior.*

Proof: Let $2a$ = the run time . Let $a + b$ the run length of the first phase of the first method, and $a + c$ be the run time of the first phase of the second method. Then the second phases have times of $a - b$ and $a - c$ respectively. Assume $b < c$.

Since the total run time is the same for both solutions, we have $Pr_1\{correct\} = Pr_2\{correct\} = e^{-\lambda 2a}$, so we need only show that $Pr_1\{detected\} < Pr_2\{detected\}$, ie.

$$
\begin{aligned}
e^{-\lambda(a+b)}(1-e^{-\lambda(a-b)}) + e^{-\lambda(a-b)}(1-e^{-\lambda(a+b)}) &< e^{-\lambda(a+c)}(1-e^{-\lambda(a-c)}) + e^{-\lambda(a-c)}(1-e^{-\lambda(a+c)}) \\
e^{-\lambda(a+b)} + e^{-\lambda(a-b)} &< e^{-\lambda(a+c)} + e^{-\lambda(a-c)} \\
e^{-\lambda b} + e^{\lambda b} &< e^{-\lambda c} + e^{\lambda c}
\end{aligned}
$$

Setting $x = e^{\lambda b}$ and $y = e^{\lambda c}$ we want

$$
\begin{aligned}
x + \frac{1}{x} &< y + \frac{1}{y} \qquad \text{for } 1 \le x < y \\
\frac{1}{x} - \frac{1}{y} &< y - x \\
\frac{y-x}{xy} &< y - x
\end{aligned}
$$

Corollary 1 *Given a basic algorithm for a problem, a certification trail method is superior to running the basic algorithm twice if the total run time is no greater than twice that of the basic algorithm.*

The above statements apply to the situation of a single execution of a solution. A more interesting case is to iterate the solution until no errors are reported, that is we either arrive at the correct answer, or have undetected errors.

Let $Pr_{iter}\{correct\}$ denote the probability of finding a correct solution in the iterated scheme and $Pr_{iter}\{undetected\}$ denote the probability of accepting an incorrect run.

Note that we repeat a run only when errors are detected, so if we obtain the correct answer on the $n-th$ run, the previous $n-1$ runs must have resulted in detected errors. Thus it is clear that:

$$
\begin{aligned}
Pr_{iter}\{correct\} &= Pr\{correct\} \sum_{i=0}^{\infty} Pr\{detected\} \\
&= \frac{Pr\{correct\}}{1 - Pr\{detected\}}
\end{aligned}
$$

Similarly,

$$
Pr_{iter}\{undetected\} = \frac{\Pr\{undetected\}}{1 - Pr\{detected\}}
$$

For the iterated scheme, we will say that one method is superior to another if the probability of obtaining the correct answer is larger. Obviously if a method is superior in the single run sense, it must be superior in the iterated case. However it is possible for one method to be superior to another in the iterated scheme, but not in the single run scheme. This means that a certification trail method may be better than running a basic algorithm twice, even if the certification trail takes longer to run!

Suppose we have a basic algorithm A with running time a for a particular problem, and a certification trail method with phases running in times b and c. Given b, how small must c be, for the certification trail to be superior?

We require:

$$\begin{aligned}
\frac{Pr_{cert}\{correct\}}{1 - Pr_{cert}\{detected\}} &> Pr_{basic}\{correct\}1 - Pr_{basic}\{detected\} \\
\frac{e^{-\lambda(b+c)}}{1 - e^{-\lambda b} - e^{-\lambda c} + 2e^{-\lambda(b+c)}} &> \frac{e^{-\lambda 2a}}{1 - 2e^{-\lambda a} + 2e^{-\lambda 2a}} \\
e^{-\lambda(b+c)} - 2e^{-\lambda(a+b+c)} &> e^{-\lambda 2a} - e^{-\lambda(2a+b)} - e^{-\lambda(2a+c)} \\
e^{-\lambda c}(e^{-\lambda b} + e^{-\lambda 2a} - 2e^{-\lambda(a+b)}) &> e^{-\lambda 2a}(1 - e^{-\lambda b})
\end{aligned}$$

Note that $b > a$, so $e^{-\lambda b} + e^{-\lambda 2a} - 2e^{-\lambda(a+b)}$ must be positive. So,

$$\begin{aligned}
e^{-\lambda c} &> \frac{e^{-\lambda 2a}(1 - e^{-\lambda b})}{e^{-\lambda b} + e^{-\lambda a}(1 - e^{-\lambda b})} \\
c &< -\frac{1}{\lambda} \ln \frac{e^{-\lambda 2a}(1 - e^{-\lambda b})}{e^{-\lambda b} + e^{-\lambda 2a}(1 - e^{-\lambda b})}
\end{aligned}$$

Since the argument to ln is strictly between 0 and 1, c is well defined for any choice of a, b, and λ.

In addition to the probability of correctness, we would like to know the expected running time using the iterated approach. Fortunately, this is easily determined.

Our probability of stopping on a particular execution is $Pr\{correct\} + Pr\{undetected\} = 1 - Pr\{detected\}$. Therefore with that probability we stop on the first execution, with probability $Pr\{detected\}(1 - Pr\{detected\})$ we stop on the second execution, and in general we stop on the nth execution with probability $(1 - Pr\{detected\})(Pr\{detected\})^{n-1}$. This gives us an expected number of iterations of,

$$(1 - Pr\{detected\}) \sum_{i=0}^{\infty} (i+1) Pr\{detected\}^i$$

Now,

$$\sum_{i=0}^{\infty} (i+1) x^i = \frac{1}{(1-x)^2}$$

so we find that the expected number of iterations is,

$$\frac{1}{1 - Pr\{detected\}}$$

Multiplying the run time of a single iteration will give us the expected running time.

Table 16 shows information for running a basic algorithm. The run time of a basic algorithm is set to 1 unit of time. The basic algorithm is run twice and the results compared, we assume that comparator is fast enough so that the time it takes is negligible (this is justified by the experimental results), and that it is error free. We compute

i. Prob. Correct - The probability that both phases are error free.

ii. Prob. Detected - The probability that exactly on of the phases contains an error.

iii. Prob. Undetected - The probability that both of the phases contain errors.

iv. Iterated Prob Correct - If the basic algorithm is iterated (each iteration is two runs), this is the probability that the terminating result is correct.

v. Expected Runtime - The expected run time of the algorithm in the iterated model. For the basic algorithm this is twice the expected number of iterations.

Tabel 17 illustrates the "breakeven" point for the certification trail approach. Given a value for λ and a run time b of a trail generating algorithm. The breakeven point for the run time of the trail checking algorithm is the

λ	Basic Algorithm	Prob Correct	Prob. Detected	Prob. Undetected	Iter. Prob. Correct	Expected Runtime
0.01	1	0.980199	0.019702	0.000099	0.999899	2.040197
0.10	1	0.818731	0.172213	0.009056	0.989060	2.416081
1.00	1	0.135335	0.465088	0.399576	0.253005	3.738935

Table 16: Balanced Probabilites

λ	Generate Trail	Breakeven Trail Checker
0.01	1.10	0.909050
0.01	1.50	0.666111
0.01	2.00	0.498750
0.10	1.10	0.908683
0.10	1.50	0.661128
0.10	2.00	0.487505
1.00	1.10	0.905504
1.00	1.50	0.614107
1.00	2.00	0.379885

Table 17: Certification checker breakeven points

point at which the iterated probability of correctness is the same as for the "basic" algorithm (which has a run time of 1).

Run times less than this will result in the certification trail solution being superior. It is interesting to notice that in the total length of the solution at the breakeven point is greater than 2, ie. running the basic algorithm twice.

Table 18 is similar to the first one, the difference being that this examines the behavior of certification trail methods for different run times of the two phases. The meaning of the other columns is identical to the meaning in the table for basic algorithms. Of interest is the row $\lambda = 1.00, b = 1.50, c = 0.25$. Compare this with the first table for $\lambda = 1.00$. We see that the certification method has a greater probability of being correct for a single run and the total run time is shorter than twice the basic algorithm, yet the expected iterated run time is larger!

10 Fault Injection Experiments

A series of hardware fault injection experiments have been conducted during which combinations of the address, data, and control lines of a Motorola

λ	Generate Certif.	Use Certif.	Prob Correct	Prob. Detected	Prob. Undetected	Iter. Prob. Correct	Expected Runtime
0.01	1.10	0.25	0.986591	0.013382	0.000027	0.999972	1.368311
0.01	1.10	0.50	0.984127	0.015818	0.000055	0.999945	1.625716
0.01	1.10	0.75	0.981670	0.018248	0.000082	0.999917	1.884387
0.01	1.50	0.25	0.982652	0.017311	0.000037	0.999962	1.780827
0.01	1.50	0.50	0.980199	0.019727	0.000074	0.999924	2.040248
0.01	1.50	0.75	0.977751	0.022138	0.000111	0.999886	2.300937
0.01	2.00	0.25	0.977751	0.022199	0.000049	0.999949	2.301082
0.01	2.00	0.50	0.975310	0.024591	0.000099	0.999899	2.563028
0.01	2.00	0.75	0.972875	0.026977	0.000148	0.999848	2.826245
0.10	1.10	0.25	0.873716	0.123712	0.002572	0.997065	1.540590
0.10	1.10	0.50	0.852144	0.142776	0.005080	0.994074	1.866490
0.10	1.10	0.75	0.831104	0.161369	0.007527	0.991025	2.205976
0.10	1.50	0.25	0.839457	0.157104	0.003439	0.995920	2.076175
0.10	1.50	0.50	0.818731	0.174476	0.006793	0.991771	2.422703
0.10	1.50	0.75	0.798516	0.191419	0.010065	0.987553	2.782653
0.10	2.00	0.25	0.798516	0.197008	0.004476	0.994426	2.802021
0.10	2.00	0.50	0.778801	0.212359	0.008841	0.988776	3.174033
0.10	2.00	0.75	0.759572	0.227330	0.013098	0.983049	3.559087
1.00	1.10	0.25	0.259240	0.593191	0.147568	0.637254	3.318513
1.00	1.10	0.50	0.201897	0.535609	0.262495	0.434755	3.445370
1.00	1.10	0.75	0.157237	0.490763	0.352000	0.308770	3.632888
1.00	1.50	0.25	0.173774	0.654383	0.171843	0.502793	5.063409
1.00	1.50	0.50	0.135335	0.558990	0.305674	0.306876	4.535047
1.00	1.50	0.75	0.105399	0.484698	0.409903	0.204539	4.366374
1.00	2.00	0.25	0.105399	0.703338	0.191263	0.355283	7.584379
1.00	2.00	0.50	0.082085	0.577696	0.340219	0.194374	5.919905
1.00	2.00	0.75	0.063928	0.479846	0.456226	0.122902	5.286897

Table 18: Unbalanced Probabilites

C-2

M68000-based target system were pulsed with selected signals of various types and durations while in the process of executing algorithms. In addition to the MC68000 microprocessor which served as the cpu, the target also was comprised of 512K bytes of RAM, 512 bytes of ROM, and numerous I/O modules to support serial and parallel communication. A timer module is also included in the target which uses the 4Mhz clock as a reference so as to provide execution time data for experiments. Finally, a simple operating system is resident in the ROM of the target which provides programming and operational support.

The fault injection testbed on which these experiments were performed is illustrated as the configuration shown in Figure 3. In addition to the target system, the fault injection testbed contains other modules which perform the fault injection and data acquisition functions under instruction from the Operations Control Console. By means of RS232C, SCSI, and GPIB interfaces, a Macintosh IICX serves as the Operations Control Console permitting fault injections to be precisely executed and resulting error data to be recorded for later analysis by a SUN SPARCstation 2.

The Operations Control Console also communicates over a VMEbus with the Testbed Controller which is responsible for overall testbed operation. The primary component of the Testbed Controller is a MC68030-based unit with 8 Mbytes of SRAM to store error data from fault injection runs as communicated to it over the VMEbus from the data acquisition module. The Testbed Controller also is similarly responsible for the operations of the fault injection module as determined by commands from the Operations Control Console.

The fault injection module and the data acquisition module have access via edge connector pins to the lines of the target system selected for injection and monitoring, respectively. The fault injections are precisely triggered after some operator determined delay following the appearance of an operator pre-selected set of bits on either the address lines of the address bus or the data lines of the data bus. Similarly, the durations and frequencies of the injections are also controlled by the operator. The injections emanate from a bank of programmable function generators included in the fault injection module. The precision with which fault conditions are triggered and injected permits the resulting error conditions which are observed to be repeated (if necessary) for further monitoring/analysis. The data acquisition module is also triggered by the same address or data bits that activated the fault injection module. However, there is no delay associated with the data acquisition function; transfer of the signals on the lines being monitored by the data

acquisition module to the memory of the Testbed Controller commences immediately the data acquisition module's activation. Data monitored by the data acquisition module is transmitted directly onto VME bus and then written into the SRAM of the Testbed Controller.

10.1 Fault injection and error classification in MC68000 target system

To generally indicate the details of the fault injection experiments using the target system, the injections and resulting errors can be summarized and displayed at the Operations Control Console as illustrated in Figure 4.

In the example illustrated in Figure 4, the trigger address for the injection was selected by the operator to be address 1019E (hexadecimal) in the first version of Huffman tree program which was to generate both the output and the certification trail. The actual injection consisted of holding the lower 4 bits of the data bus at logical zero starting 2 microseconds after the recognition of the trigger address by the fault injection module and then maintaining the logical zero on these lines for various durations lasting between 1 and 10 microseconds. For this example, we see that 5 distinct error conditions resulted depending on the duration of the injection. The details of data errors classified as type 2 and type 3 are beyond the scope of this discussion. Suffice it to say that each such type of data error observed in this particular experimental run could be interpreted as an inconsistent labeling of nodes in the certification trail passed to the second program. In each case, however, it should be emphasized that the execution of the second program utilizing the certification trail detected the error. The other errors listed in Figure 4 can be categorized as address errors and illegal instructions.

Our purpose in presenting Figure 4 is only to illustrate an example of a fault injection run with a subsequent error analysis and classification. In general, the errors resulting from injections into the target system could be classified as:

- No error.
- Data output errors
- Certification trail errors
- Addressing errors
- Data value errors

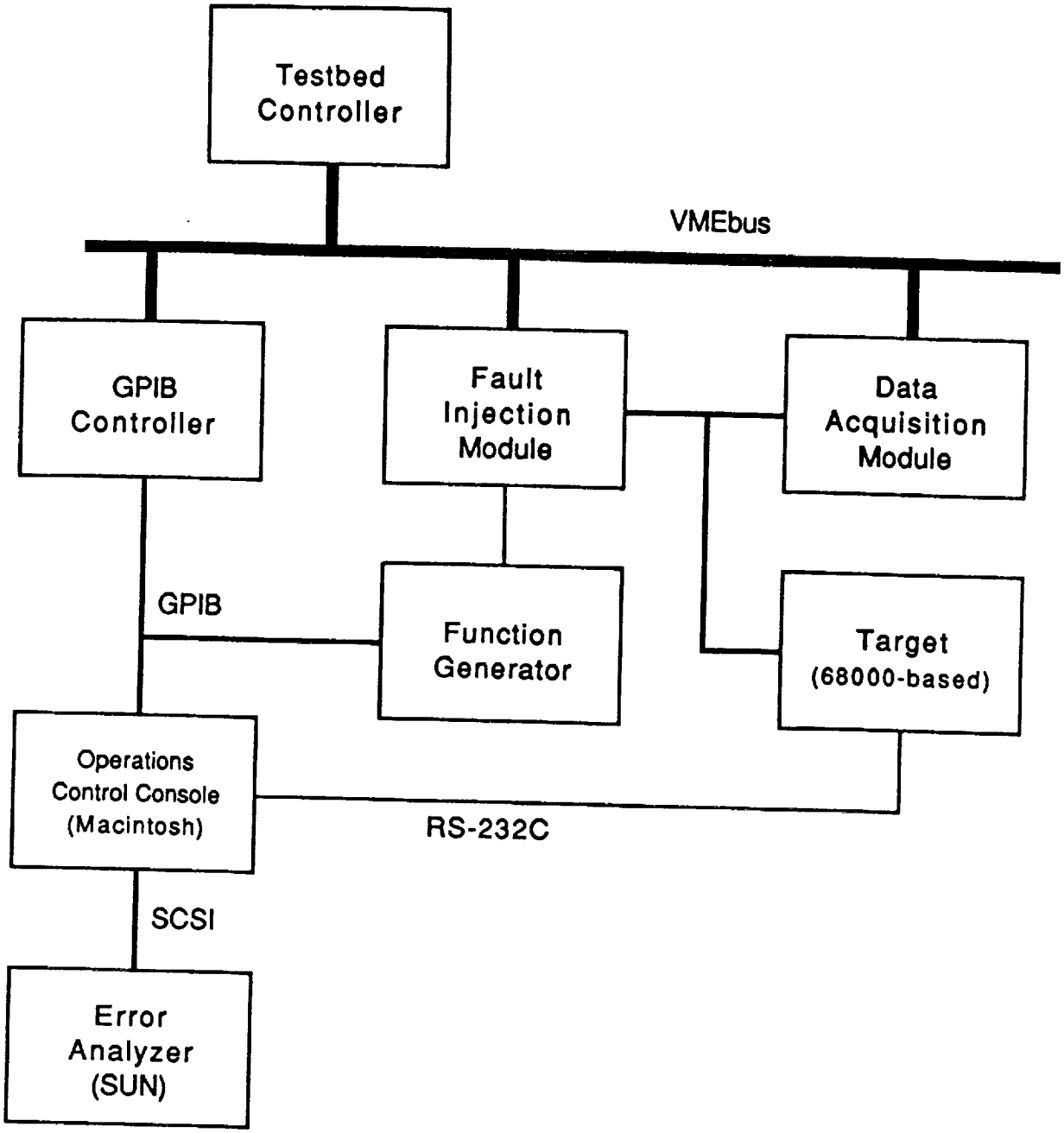


Figure 3: Hardware fault injection testbed for MC68000-based target system

Fault	Delay	Width	Error
XXXX XXX0	0 us	.1 us	no error
		.2	no error
		.3	no error
		.4	ADDR TRAP ERROR
		.5	ADDR TRAP ERROR
		1	ADDR TRAP ERROR
		2	ADDR TRAP ERROR
		4	ADDR TRAP ERROR
		4.5	ADDR TRAP ERROR
		5	data_error.2
		5.5	Certification Error: Inconsistent Labels data_error.2
		6	Certification Error: Inconsistent Labels data_error.3
		7	Certification Error: Inconsistent Labels data_error.3
		8	Certification Error: Inconsistent Labels data_error.3
		9	Certification Error: Inconsistent Labels data_error.3
		10	Certification Error: Inconsistent Labels ILLEGAL INSTRUCTION

Figure 4: Example of output displayed at Operations Control Console for fault injection run for Huffman tree algorithm program

- Halt generated
- Reset generated
- Non-termination of program
- Program mutilation

Currently, the testbed tools are being expanded to produce automated injections using suites of fault conditions on the target system.

Software fault injection experiments were also performed in which instructions, data, and stack contents were modified using both the Sun Sparcstation and the 386 machine with which the previously detailed timing data was collected. The details of these fault injection experiments will be presented in a companion document.

11 Concluding Discussion

This paper experimentally supplements two previous *FTCS* papers [1, ?] which theoretically explore the new fault tolerance technique referred to as the certification trail method. We have presented experimental timing data which illustrates the advantages of the certification trail technique over classical time redundancy. We have further presented analytical results which further support the significance of the certfication trail technique.

References

[1] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[2] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Department of Computer Science Technical Report JHU 89/26*, Johns Hopkins University, Baltimore, Maryland, 1989.

[3] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Digest of the 1991 Fault Tolerant Computing Symposium*, pp. 240-247, IEEE Computer Society Press, 1991.

[4] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Department of Computer Science Technical Report JHU 90/17*, Johns Hopkins University, Baltimore, Maryland, 1990.

[5] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[6] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[7] Andrews, D., "Software fault tolerance through executable assertions," *Rec. 12th Asilomar Conf. Circuits, Syst., Comput.*, pp. 641-645, 1978, Nov. 6-8.

[8] Andrews, D., "Using excutable assertions for testing and fault tolerance," *Dig. 9th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 102-105, 1979, June 20-22.

[9] Avizienis, A., "Fault tolerance by means of external monitoring of computer systems," *Proceedings of the 1981 National Computer Conference*, pp. 27-40, AFIPS Press, 1980

[10] Avizienis, A., "Design diversity - the challenge of the eighties," *Digest of the 1982 Fault Tolerant Computing Symposium*, pp. 44-45, IEEE Computer Society Press, 1982.

[11] Avizienis, A., and Kelly, J., "Fault tolerance by design diversity: concepts and experiments," *Computer*, vol. 17, pp. 67-80, Aug., 1984.

[12] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[13] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[14] Blough, D., and Masson, G., "Performance analysis of a generalized concurrent error detection procedure," *IEEE Trans. on Computers* vol. 39, Jan., 1990.

[15] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[16] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[17] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[18] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, Sept., 1959.

[19] Eifert, J.B., and Shen, J.P., "Processor monitoring using asynchronous signatured instruction streams," *Dig. 14th Int. Conf. Fault-Tolerant Comput.*, pp. 394-399, 1984, June 20-22.

[20] Fredman, M. L., and Willard, D. E., "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st IEEE Foundations of Computer Science*, pp. 719-725,1990.

[21] Fredman, M. L., and Saks, M. E., "The cell probe complexity of dynamic data structures," *Proc. 21st ACM Symp. on Theo. Comp. 1989*, pp. 109-122, 2, 1986.

[22] Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E., "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica 6*, pp. 109-122, 2, 1986.

[23] Gabow, H. N., and Tarjan, R. E., "A linear-time algorithm for a special case of disjoint set union," *J. of Comp. and Sys. Sci.*, 30(2), pp. 209-221, 1985.

[24] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set", *Information Processing Letters*, pp. 132-133, 1, 1972.

[25] Graham, R. L., and Hell, P., "On the history of the minimum spanning tree problem," *Ann. Hist. Comput.*, pp. 43-47, Jan., 1985.

[26] Hoare, C. A. R., "Quicksort," *Computer Journal*, pp. 10-15, 5(1), 1962.

[27] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[28] Gunneflo, U., Karlsson, J., and Torin, J., "Evaluation of error detection schemes for using fault injection by heavy-ion radiation," *Dig. of the 1989 Fault Tolerant Computing Symposium*, pp. 340-347, June, 1989.

[29] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

[30] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[31] Iyengar, V.S. and Kinney, L.L., "Concurrent fault detection in microprogrammed control units," *IEEE Trans. Comput.*, vol. C-34, pp. 810-821, Sept. 1985.

[32] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[33] "Fault tolerant FFT networks," *Dig. of the 1985 Fault Tolerant Computing Symposium*, June, 1985.

[34] Kane, J.R. and Yau, S.S., "Concurrent software fault detection," *IEEE Trans. Software Eng.* , vol. SE-1, pp. 87-99, March 1975.

[35] Komlòs, J., "Linear verification for spanning trees", *Proceedings of the 1984 Symposium on Foundations of Computing*, pp. 201-206, IEEE Computer Society Press, 1984.

[36] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[37] Lu, D., "Watchdog processor and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, pp. 681-685, July 1982.

[38] Mahmood, A., Lu, D.J. and McCluskey, E.J., "Concurrent fault detection using a watchdog processor and assertions," *Proc. 1983 Int. Test Conf.*,, pp. 622-628, Oct., 1983.

[39] Mahmood, A. Ersoz, A. and McCluskey, E.J., "Concurrent system level error detection using a watchdog processor," *Proc. 1985 Int. Test Conf.*, pp. 145-152, Nov., 1985.

[40] Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors - a survey," *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb., 1988.

[41] Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors", *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb., 1988.

[42] Nair, V., and Abraham, J., "General linear codes for fault-tolerant matrix operations on processor arrays," *Dig. of the 1988 Fault Tolerant Computing Symposium*, pp. 180-185, June, 1988.

[43] Namjoo, M., and McCluskey, E., "Watchdog processors and capability checking," *Digest of the 1982 Fault Tolerant Computing Symposium*, pp. 245-248, IEEE Computer Society Press, 1982.

[44] Namjoo, M. "Techniques for concurrent testing of VLSI processor operation," *Dig. 1982 Int. Test Conf.*, pp. 461-468, Nov., 1982.

[45] Namjoo, M. "CERBERUS-16: An architecture for a general purpose watchdog processor," *Dig. Papers 13th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 216-219, June, 1983.

[46] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

[47] Prim, R. C., "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, pp. 1389-1401, Nov., 1957.

[48] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[49] Schmid, M., Trapp, R., Davidoff, A., and Masson, G., "Upset exposure by means of abstraction verefication," *Dig. of the 1982 Fault Tolerant Computing Symposium*, pp. 237-244, June, 1982.

[50] Sedgewick, R., "Implementing quicksort programs," *Communications of the ACM*, pp. 847-857, 21(10), 1978.

[51] Shen, J.P. and Schuette, M.A., "On-line self-monitoring using signatured instruction streams," *Proc. 1983 Int. Test Conf.*, pp. 275-282, Oct., 1983.

[52] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[53] Sridhar, T. and Thatte, S.M., "Concurrent checking of program flow in VLSI processors," *Dig. 1982 Int. Test Conf.*, pp. 191-199, Nov., 1982.

[54] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[55] Tarjan, R. E., "Efficiency of a good but not linear set union algorithm," *J. ACM*, 22(2), pp. 215-225, 1975.

[56] Tarjan, R. E., "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. of Comp. and Sys. Sci.*, 18(2), pp. 110-127, 1979.

[57] Tarjan, R. E., and Leeuwen, J. van, "Worst-case analysis of set union algorithms," *J. ACM*, 31(2), pp. 245-281, 1984.

[58] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[59] Tomas, S. P. and Shen, J. P., "A roving monitoring processor for detection of control flow errors in multiple processor systems," *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput.*, pp.531-539, Oct., 1985.

[60] Taylor, D., "Error Models for robust data structures," *Dig. 20th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 416-422, 1990 June 26-28.

[61] Williams, J. W. J, "Algorithm 232 (heapsort)," *Commun. of ACM*, vol.7, pp. 347-348, 1964.

[62] Yau, S.S, and Chen, F.-C., "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126-137, March 1980.

APPENDIX A

DATA ACQUISITION MODULE TECHNICAL MANUAL Ver. 1.0

THE TABLE OF CONTENTS

1. The Experimental System Overview

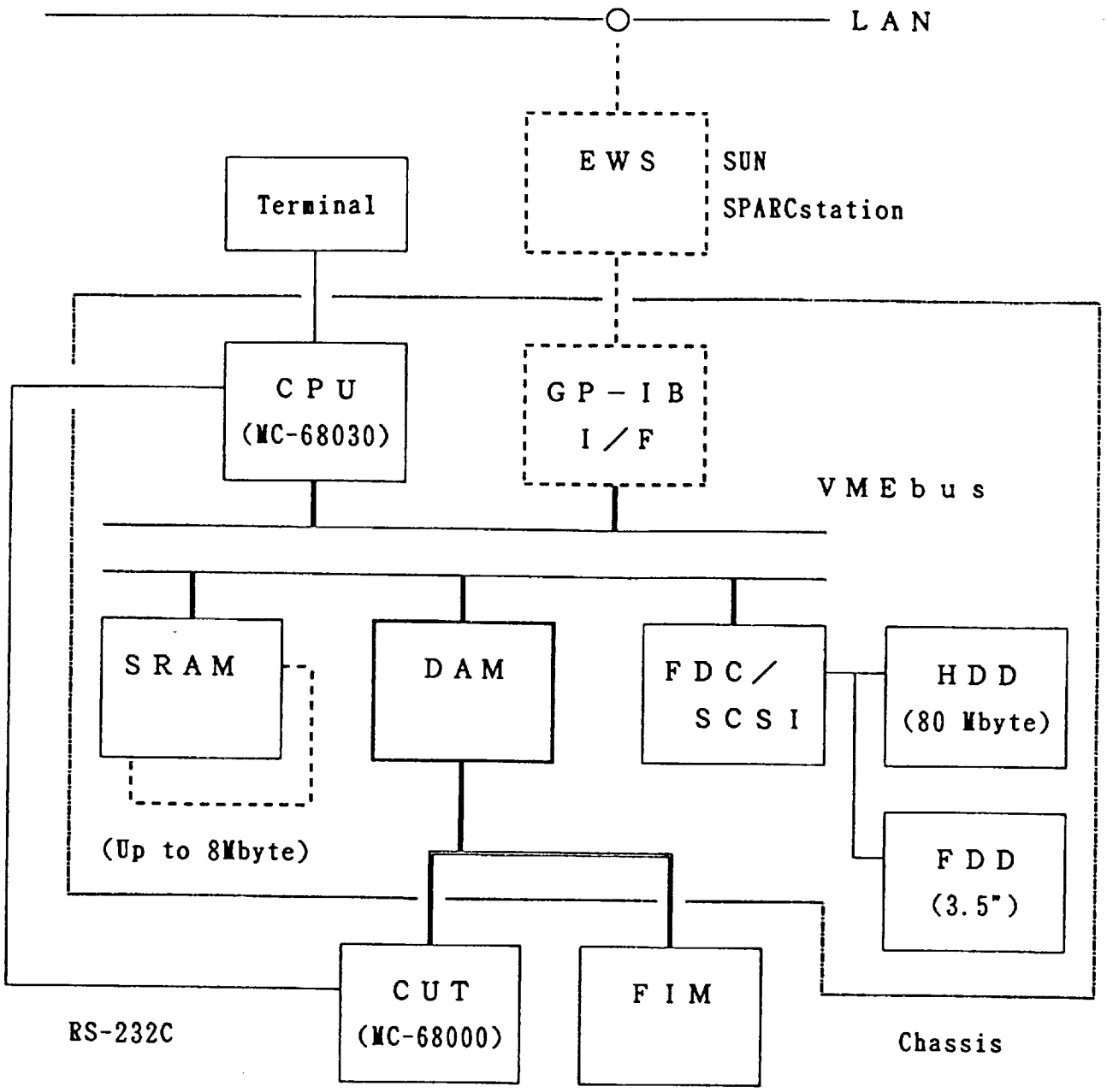
This system provides an experimental environment for recording and analyzing upset data in computer systems. This chapter provides the information on the system configuration and general hardware description.

1.1 System Configuration

This experimental system is mainly based on the VMEbus and controlled by the 68030 CPU board. The VMEbus provides a master-slave, asynchronous non- multiplexed data transfer medium. The target system (CPU Under Test) and the Fault Injection Module are connected by its local bus.

Fig.1.1 shows the experimental configuration. This system's features include:

- 68030 CPU Board
- Up to 8 Mbyte SRAM Memory Modules
- Floppy Disk and SCSI Bus Controller (FDC/SCSI)
- 80 Mbyte Hard Disk and 3.5" Floppy Disk Drive
- OS-9 Operating System
- Chassis with power supply, cooling fans, and motherboard
- Data Acquisition Module
- CPU Under Test (MC68000 Educational Computer Board)
- Fault Injection Module
- (GP-IB I/F Controller)
- (SUN SPARCstation)



DAM: Data Acquisition Module
CUT: CPU Under Test (Target System)
FIM: Fault Injection Module

Fig. 1.1 Experimental Configuration

1.2 General System Description

This section briefly describes the general description of each module of the experimental system. For detailed information, refer to the user's manuals on specific modules.

- 68030 CPU Board
  - SYS68K/CPU-33XN (Force Computers Inc.)
  - 68030 CPU with 16.7 MHz clock frequency.
  - Not equipped with the Floating Point Coprocessor.
  - 32-bit high speed DMA controller for data transfers.
  - 1 Mbyte of shared dynamic RAM.
  - Two multiprotocol serial I/O channels.
  - Up to 2 Mbyte EPROM and up to 512 Kbyte SRAM/EEPROM.
  - Real Time Clock with calendar and on-board battery backup.
  - Full 32 bit VMEbus master/slave interface.
- Memory Module
  - SYS68K/SRAM-6 (Force Computers Inc.)
  - 2 Mbyte SRAM on SRAM-6.
  - Battery backup for SRAM devices.
  - 55ns(typical) Read/Write Access Time.
  - Jumper selectable access address and address modifier code.
  - VMEbus intereface supporting 32 data and 32 address lines.
- Floppy Disk and SCSI Bus Controller
  - SYS68K/ISCSI-1 (Force Computers Inc.)
  - 68010 CPU for local control.
  - 68450 DMA Controller for local transfers.
  - SCSI bus interface with the NCR5386S SCSI bus controller.

- SHUGART compatible floppy interface with the WD1772 FDC.
- All I/O signals available on P2 connector.
- VMEbus interface supporting A24:D16, D8.

- Mass Storage Module
  - SYS68K/MSM-84 (Force Computers Inc.)
  - Only VME P1 backplane is required.
  - 64 Pin flat cable is used to connect P2 of the ISCSI-1.
  - Floppy Disk Driver (Toshiba ND352)
    * Disk Size and Capacity: 3.5", 1.0 Mbyte
    * Number of Tracks: 160
    * Access Time: 79 ms (average)
  - Hard Disk (Quantum PRO80S)
    * Disk Size and Capacity: 3.5", 84 Mbyte
    * Number of Cylinders and Heads: 834, 6
    * Seek Time: 19 ms (average)

- OS-9 Operating System
  - Professional OS-9 (Microware Systems Corporation)
  - Multitasking, real time operating system.
  - UNIX-like shell and a hierarchical directory/file structure.
  - C Compiler, Assembler/Linker, and User-state Debugger.
  - μMACS screen-oriented text editor.

- Chassis with power supply, cooling fans, and motherboard
  - SYS68K/TARGET-32 (Force Computers Inc.)
  - 19", 7U chassis.
  - 500 W power supply to drive VMEbus and mass storage memory.
  - Cooling systems with four fans.
  - 20 slot J1-J2 VMEbus Motherboard.

- Data Acquisition Module
  - Up to 8 Mbyte address space.
  - Jumper selectable address modifier code.
  - 32 Input Channels with data selectors.
  - VMEbus compatible data transfers supporting A24:D32, D8.
  - VMEbus Master bus control (Non-slot 1)
- CPU Under Test
  - MC68000 Educational Computer Board (Motorola Inc.)
  - 4 MHz MC68000 16-bit CPU.
  - 32 Kbyte of DRAM and 16 Kbyte firmware ROM/EPROM monitor.
  - Two serial ports provided for a terminal and a host.
- Fault Injection Module
  - Hardware fault injections on IC pin lines.
  - Single/multiple faults of stuck/bridging types with fault duration varying from 250 ns to ~~ms~~ 64 μs.
  - Application program generated fault injection.

1.3 System Customization

This section describes the system customization required to implement the upset analysis experimental system. This also provides information on the programming of peripherals.

- SYS68K/CPU-33XN
  - OS-9/68000[1] EPROM Installation
    - Remove VMEPROM[2] and install EPROMs for OS-9.
    - High — Socket J6, Low — Socket J4
  - EPROM Type Selection
    - 27512 EPROM
    - Jumperfield B1: 1 to 12, 6 to 7
  - Interfacing PI/T2 User I/O Port
    - Device: MC68230 Parallel Interface/Timer (PI/T)
    - Accessible via the 8-bit local I/O bus. Table 1.1 shows the register layout of PI/T2.
    - User I/O port is available on P2 of VMEbus, shown in Table 1.2.
  - The Address Map
    - The address map of this CPU board is listed in Table 1.3.
    - A24: D32, D24, D16, D8 area: SRAM-6, ISCSI-1
- SYS68K/SRAM-6
  - Address Modifier Selection
    - Standard Supervisor/Non-privileged Data Access
    - Address Modifier Code: 3D, 39
    - Jumperfield B4: 4 to 15, 2 to 17
  - VMEbus Interface
    - A24: D32, D16, D8
    - Standard Address Mode (A24)

    * Address: $XX000000 — $XX2000000 (2 Mbyte)
    * Jumperfield B3: 18 to 15, 20 - 30 to 13 - 3

- SYS68K/ISCSI-1
  - Address Modifier Selection
    * Standard Non-priviledged/Supervisory program and data Access.
    * Address Modifier Code: 3A, 39, 3E, 3D
    * Jumperfield B22: 5 to 2, 6 to 1
  - VMEbus Interface
    * A24: D16, D8
    * Address: $XXA00000 — $XXA1FFFF (128 Kbyte)
    * Jumperfield B21: 2 to 17, 4 - 7 to 15 - 12

Table 1.1 PI/T2 Register Layout

ADDRESS	REGISTER	DESCRIPTION
FF800E00	PIT2 PGCR	Port General Control Register
FF800E01	PIT2 PSRR	Port Service Request Register
FF800E02	PIT2 PADDR	Port A Data Direction Register
FF800E06	PIT2 PACR	Port A Control Register
FF800E08	PIT2 PADR	Port A Data Register
FF800E0A	PIT2 PAAR	Port A Alternate Register
FF800E0D	PIT2 PSR	Port Status Register

Table 1.2 PI/T2 User I/O Interface Signals

PIN No.	PORT No.	IN/OUT	P2/J2 No.	SIGNAL
4	PA0	OUT	A29	READY*
5	PA1	OUT	C29	LW/B*
6	PA2	OUT	A30	SLCT0*
7	PA3	OUT	C30	SLCT1*
8	PA4	IN	A31	ENB0*
9	PA5	IN	C31	ENB1*
10	PA6		A32	
11	PA7		C32	
13	H1		A27	
14	H2		C27	
15	H3		A28	
16	H4		C28	

Table 1.3 The Address Map

START (HEX)	END (HEX)	SPACE	DESCRIPTION
00000000	003FFFFF	1.0 MB	Shared Memory
00400000	F9FFFFFF	3.9 GB	A32: D32, D24, D16, D8
FA000000	FAFFFFFF	16.0 MB	Message Broadcast Area
FB000000	FBFEFFFF	15.9 MB	A24: D32, D24, D16, D8
FBFF0000	FBFFFFFF	64.0 KB	A16: D32, D24, D16, D8
FC000000	FCFEFFFF	15.9 MB	A24: D16, D8
FCFF0000	FCFFFFFF	64.0 KB	A16: D16, D8
FD000000	FFFFFFFF		System Area

[1]OS-9 and OS-9/68000 are trademarks of Microware Systems Corporation.
[2]VMEPROM is a PDOS based real time monitor.

2. Data Acquisition Module

When the fault is injected from the fault injection module, the data acquisition module is activated and activity data on 8 or 32 observation points are synchronously sampled with the clock of the target system and written into the SRAM memory module.

2.1 Hardware Overview

Basically, the data acquisition module generates the address signals from the clock of the target system and transfers the sampled data to the memory module via the VMEbus.

A block diagram is shown in Fig.2.1. This board consists of the following functional blocks:

- Clock Control (CKCTRL)
- Address Generator (ADDGEN)
- Address Modifier Selector (AMS)
- Address Bus Buffers (ABUF)
- Data Transfer Control (DTCTRL)
- Input Channel Selectors (INSLCT)
- Data Bus Buffers (DBUF)
- Bus Master Control (BUSMST)

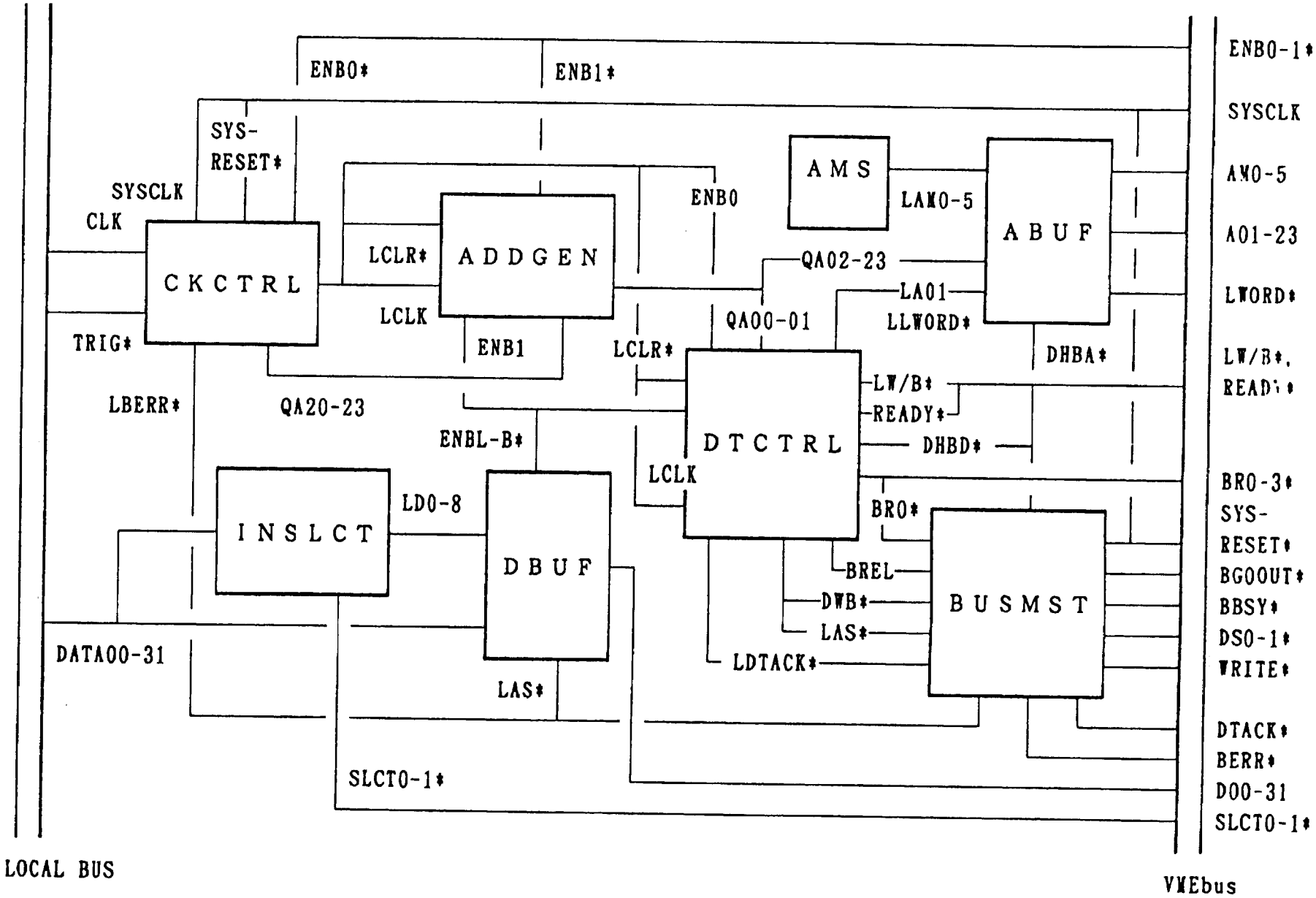


Fig. 2.1 Functional Block Diagran

2.2 Clock Control

- Recording Clock Selector
  - J1-1, IC1-1
  - Selectable by bit 1 and 2 of J1.
    * Clock of CPU Under Test: bit 1: ON, bit 2: OFF
    * 16MHz VME System Clock: bit 1: OFF, bit 2: ON
- Clock Frequency Divider
  - J1-2, IC2
  - Selectable by bit 3 – 7 of J1 as shown in Table 2.1.
- Qualifier Trigger
  - IC1-2, IC3-1, IC10-1
  - Trigger: Fault injection signal transferred from FIM.
  - The trigger is enabled when ENB1 is high.
- Clear Control
  - R1, IC1-3, IC16-1
  - Generate Clear Signal for the Clock Control, Address Generator, and Data Transfer Control.
  - Reset Signals: System Reset, Bus Error, and End Address.
- End Address Selection
  - J2-1
  - End address: $XX0FFFFF – $XX7FFFFF
  - Selectable by bit 1 – 4 of J2-1 as shown in Table 2.2.

Table 2.1 Frequency Division Settings

Division	bit 3	bit 4	bit 5	bit 6	bit 7
1	ON	OFF	OFF	OFF	OFF
2	OFF	ON	OFF	OFF	OFF
4	OFF	OFF	ON	OFF	OFF
8	OFF	OFF	OFF	ON	OFF
16	OFF	OFF	OFF	OFF	ON

Table 2.2 End Address Selection

End Address	bit 1	bit 2	bit 3	bit 4
$XX0FFFFF	ON	OFF	OFF	OFF
$XX1FFFFF	OFF	ON	OFF	OFF
$XX3FFFFF	OFF	OFF	ON	OFF
$XX7FFFFF	OFF	OFF	OFF	ON

2.3 Address Generator

- Address Signal Generator
  - IC4, IC5, IC6, IC7, IC8, IC9
  - Implement 24-bit synchronous binary counter using a carry-look-ahead circuit.
  - Maximum clock frequency is calculated as follows: $f_{MAX} = 1/(CLKtoRCOt_{PLH} + ENTt_{SU})$
  - Address Space
    - Up to 8 Mbyte Address Space. Refer to Table 2.3.
    - Start address: **$XX000000** (fixed)
    - End address: **$XX0FFFFF** – **$XX7FFFFF** (selectable)
- Counter Status Output
  - IC10-2
  - When counters are enabled to count, **ENB1*** is asserted.

Table 2.3 Address Space and End Address

Address Space	End Address
1 Mbyte	**$XX0FFFFF**
2 Mbyte	**$XX1FFFFF**
4 Mbyte	**$XX3FFFFF**
8 Mbyte	**$XX7FFFFF**

2.4 Address Bus Buffers and Address Modifier Selector

- Address Bus Buffers
  - IC12, IC13, IC14
  - Three transparent D-latches (74AS573) interface local address signals with the VMEbus address bus.
  - DHBA* places the 24-bit outputs in either a normal logic state or a high-impedance state.
- Address Modifier Selector
  - J2-2, RN, IC11
  - 6-bit Codes: Used for an additional decoding parallel to the address signals.
  - Address Mode: Supports the standard address mode (A24) for supervisor or nonpriviledged memory access.
    * 3E: Standard Supervisor Program Access
    * 3D: Standard Supervisor Data Access
    * 3A: Standard Non-Priviledged Program Access
    * 39: Standard Non-Priviledged Data Access
  - Selectable by bit 5 - 10 of J2 as shown in Table 2.4.

Table 2.4 Address Modifier Codes and Settings

HEX	Binary	bit 5	bit 6	bit 7	bit 8	bit 9	bit 10
3E	111110	OFF	OFF	OFF	OFF	OFF	ON
3D	111101	OFF	OFF	OFF	OFF	ON	OFF
3A	111010	OFF	OFF	OFF	ON	OFF	ON
39	111001	OFF	OFF	OFF	ON	ON	OFF

2.5 Data Transfer Control

- Data Transfer Bus Control
  - ENB1, DWB*
    - * IC10-3, IC15-1
    - * When READY* asserted, both ENB1 and DWB* are latched to be active.
    - * LCLR* resets the outputs.
  - LAS*
    - * R2, IC10-4, IC15-2, IC17-1, -2
    - * When READY* asserted, LAS* is set to be active.
    - * During data transfers, LAS* is asserted by LCLK and reset by LDTACK*.
  - LA01, LDS0–1*, LLWORD*
    - * IC16-2, -3, -4, IC18-1, -2, IC30-1, -2, -3, IC33-1
    - * When LW/B* is high (long word mode), LDS0*, LDS1*, LA01, and LLWORD* are set to low during data transfers.
    - * When LW/B* is low (byte mode), LLWORD* is set to high and other signals respond as follows:
      LDS0* = QA00, LDS1* = –QA00, LA01 = QA01
- Data Bus Buffer Control
  - IC17-3, -4, IC18-4, -5
  - Long Word Mode (LW/B* is high)
    - * During DHBD* is active, ENBL* is asserted and ENBB* is de-asserted.
  - Byte Mode (LW/B* is low)
    - * During DHBD* is active, ENBB* is asserted and ENBL* is de-asserted.

- Bus Release Control
  - IC31-1
  - Support Release On Request (ROR) operation.
    * Bus request signals (BR0–3*) will assert BREL to release BBSY* at the end of the current data transfer.

2.6 Input Channel Selector and Data Bus Buffers

- Input Channel Selector
  - IC10-5, -6, IC19, IC20, IC21, IC22
  - Implement 32-to-8 data selectors using four 4-bit data selectors.
  - Data selection is controlled by the two select inputs (SCLT0–1*) as shown in Table 2.5.
- Data Bus Buffers
  - Long Word Mode
    * IC23, IC24, IC25, IC26
    * Four transparent D-latches (74AS573) interface 32-bit input data with the 32-bit VME data bus (D00–31).
    * When LAS* is taken low, the outputs are latched to retain the data that was set up. Refer to Table 2.6.
    * ENBL* places the 32-bit outputs in either a normal logic state or a high-impedance state.
  - Byte Mode
    * IC27, IC32
    * Two transparent D-latches (74AS573) interface 8-bit local data bus (LD0–7) with the 16-bit VME data bus (D00-15).
    * When LAS* is taken low, the outputs are latched to retain the data that was set up. Refer to Table 2.6.
    * ENBB* places the 16-bit outputs in either a normal logic state or a high-impedance state.

Table 2.5 Input Channel Selection

SLCT0*	SLCT1*	LD7	LD6	LD5	LD4	LD3	LD2	LD1	LD0
high	high	28	24	20	16	12	08	04	00
high	low	29	25	21	17	13	09	05	01
low	high	30	26	22	18	14	10	06	02
low	low	31	27	23	19	15	11	07	03

Table 2.6 (a) Active Portions of Data Bus

DS1*	DS0*	A01	LWORD*	D24–31	D16–23	D08–15	D00–07
low	low	low	low	byte 0	byte 1	byte 2	byte 3
high	low	high	high				byte 3
low	high	high	high			byte 2	
high	low	low	high				byte 1
low	high	low	high			byte 0	

Table 2.6 (b) Data Organization in Memory

Operand	Byte Address
byte 0	$XXX....XX00
byte 1	$XXX....XX01
byte 2	$XXX....XX10
byte 3	$XXX....XX11

2.7 VMEbus Master Control

- Master Bus Controller
  - IC28, IC29
  - VME 1220[1] provides two device chip set for non-slot 1 master bus controller.
  - Initiating a Bus Request
    - Drive BR0* low after receiving DWB* and LAS* asserted.
  - Arbitration
    - After receiving BG0IN* from daisy chained VMEbus grants, local arbiter arbitrates between DWB* and BG0IN.
      - If DWB* wins the arbitration (i.e. DWB* occurs before BG0IN*), BBSY* will be asserted.
      - If BG0IN* wins, local arbiter will drive BG0OUT*, which passes the bus grant down the daisy chain to adjacent master in the system.
  - Data Transfer
    - Local master does not access the bus until the previous master has relinquished control of bus, which occurs when AS*, DTACK* and BERR* are de-asserted.
    - Support Address Pipelining using DHBA* and DHBD*.
      - Broadcast the address of the next bus cycle while the data transfer of the current cycle is occuring, i.e. DTACK* and DSn* are still low.
      - DHBA* is enabled as soon as AS* is disabled.
      - When DTACK* goes high, signifying the end of the current data cycle, DHBD* enables the data buffers for the next data cycle.
    - WRITE* is latched during address pipelining to hold its level.
  - Bus Release
    - Supports Release On Request (ROR) protocol via BREL.
      - Release the data transfer bus whenever another module requires it.

- External bus request will assert BREL to release BBSY* at the end of the current data transfer. Refer to section 2.5.
- If no bus requests are pending, the BREL will be kept de-asserted and the local master maintains BBSY* low to perform continuous VMEbus data transfer cycles.

[1]PLX Technology, 625 Clyde Ave., Mountain View, CA 94043

3. Interface Signals

3.1 VMEbus Interface

This section provides information on VMEbus interface. Table 3.1 and Table 3.2 list P1/J1 and P2/J2 pin assignments respectively. The P1 connector includes all the signals required for the 68000. The P2 connector provides expansion of both address and data buses to 32 bits and also provides 96 pins for user I/O lines.

The data transfer bus is very similar to the 68000's native buses except the following signals. Long word (LWORD*) is asserted for 32-bit data transfers. The 6-bit address modifier (AM0 - AM5) allows the type of access to be specified. The bus error signal (BERR*) is typically used to indicate a memory error.

The interrupt bus has seven interrupt request lines (IRQi*), an interrupt acknowledge (IACK*), and a daisy-chained priority signal (IACKIN*, IACKOUT*). Each of seven lines corresponds to an interrupt priority level.

The arbitration bus provides four levels of arbitration. For each level, there is a bus request signal (BRi*) and a bus grant daisy chain (BGiIN*, BGiOUT*). The utility bus consists of SYSCLK, SYSRESET*, SYSFAIL*, ACFAIL*, and power supplies.

Table 3.1 VMEbus P1/J1 Pin Assignments

PIN No.	P1/J1 ROW A	P1/J1 ROW B	P1/J1 ROW C
1	D00	BBSY*	D08
2	D01	BCLR*	D09
3	D02	ACFAIL*	D10
4	D03	BG0IN*	D11
5	D04	BG0OUT*	D12
6	D05	BG1IN*	D13
7	D06	BG1OUT*	D14
8	D07	BG2IN*	D15
9	GND	BG2OUT*	GND
10	SYSCLK	BG3IN*	SYSFAIL*
11	. GND	BG3OUT*	BERR*
12	DS1*	BR0*	SYSRESET*
13	DS0*	BR1*	LWORD*
14	WRITE*	BR2*	AM5
15	GND	BR3*	A23
16	DTACK*	AM0	A22
17	GND	AM1	A21
18	AS*	AM2	A20
19	GND	AM3	A19
20	IACK*	GND	A18
21	IACKIN*	SERCLK	A17
22	IACKOUT*	SERDAT*	A16
23	AM4	GND	A15
24	A07	IRQ7*	A14
25	A06	IRQ6*	A13
26	A05	IRQ5*	A12
27	A04	IRQ4*	A11
28	A03	IRQ3*	A10
29	A02	IRQ2*	A09
30	A01	IRQ1*	A08
31	−12VDC	+5VSTDBY	+12VDC
32	+5VDC	+5VDC	+5VDC

Table 3.2 VMEbus P2/J2 Pin Assignments

PIN No.	P2/J2 ROW A	P2/J2 ROW B	P2/J2 ROW C
1		+5VDC	
2		GND	
3		RESERVED	
4		A24	
5		A25	
6		A26	
7		A27	
8		A28	
9		A29	
10		A30	
11		A31	
12		GND	
13		+5VDC	
14		D16	
15		D17	
16		D18	
17		D19	
18		D20	
19		D21	
20		D22	
21		D23	
22		GND	
23		D24	
24		D25	
25		D26	
26		D27	
27		D28	
28		D29	
29	READY*	D30	LW/B*
30	SLCT0*	D31	SLCT1*
31	ENB0*	GND	ENB1*
32		+5VDC	

3.2 Input Channels

The input channels consit of data channels (DATA00–31), clock (CLK), and trigger signal (TRIG*). Table 3.3 shows the pin assignments of the input channels.

Table 3.3 Input Channel Pin Assignments

PIN	DAM Signal	ECB Signal	PIN	DAM Signal	ECB Signal
(a)			(b)	GND	GND
(c)			(d)	GND	GND
1	DATA04	D04	2	DATA03	D03
3	DATA05	D05	4	DATA02	D02
5	DATA06	D06	6	CLK	4M-CLK
7	DATA07	D07	8	DATA14	D14
9	DATA08	D08	10	DATA15	D15
11	DATA09	D09	12	TRIG*	FIEN*[1]
13	DATA10	D10	14	DATA01	D01
15	DATA11	D11	16		E
17	DATA12	D12	18		AS*
19	DATA13	D13	20		UDS*
21	DATA00	D00	22		LDS*
23	DATA31	A15	24	DATA16	R/W*
25	DATA30	A14	26	DATA29	A13
27	DATA28	A12	28		FC2
29	DATA27	A11	30		FC1
31	DATA26	A10	32		FC0
33	DATA25	A09	34	DATA17	A01
35	DATA24	A08	36	DATA18	A02
37	DATA22	A06	38	DATA19	A03
39	DATA23	A07	40	DATA20	A04
41	DATA21	A05	42		DTACK*
43		8M-CLK	44		6800IRQ*
45		1M-CLK	46		VMA*

[1]FIEN*: Fault Injection Enable, a signal transferred from the fault injection module.

Appendix A Schematic Diagrams

A.1 Clock Control

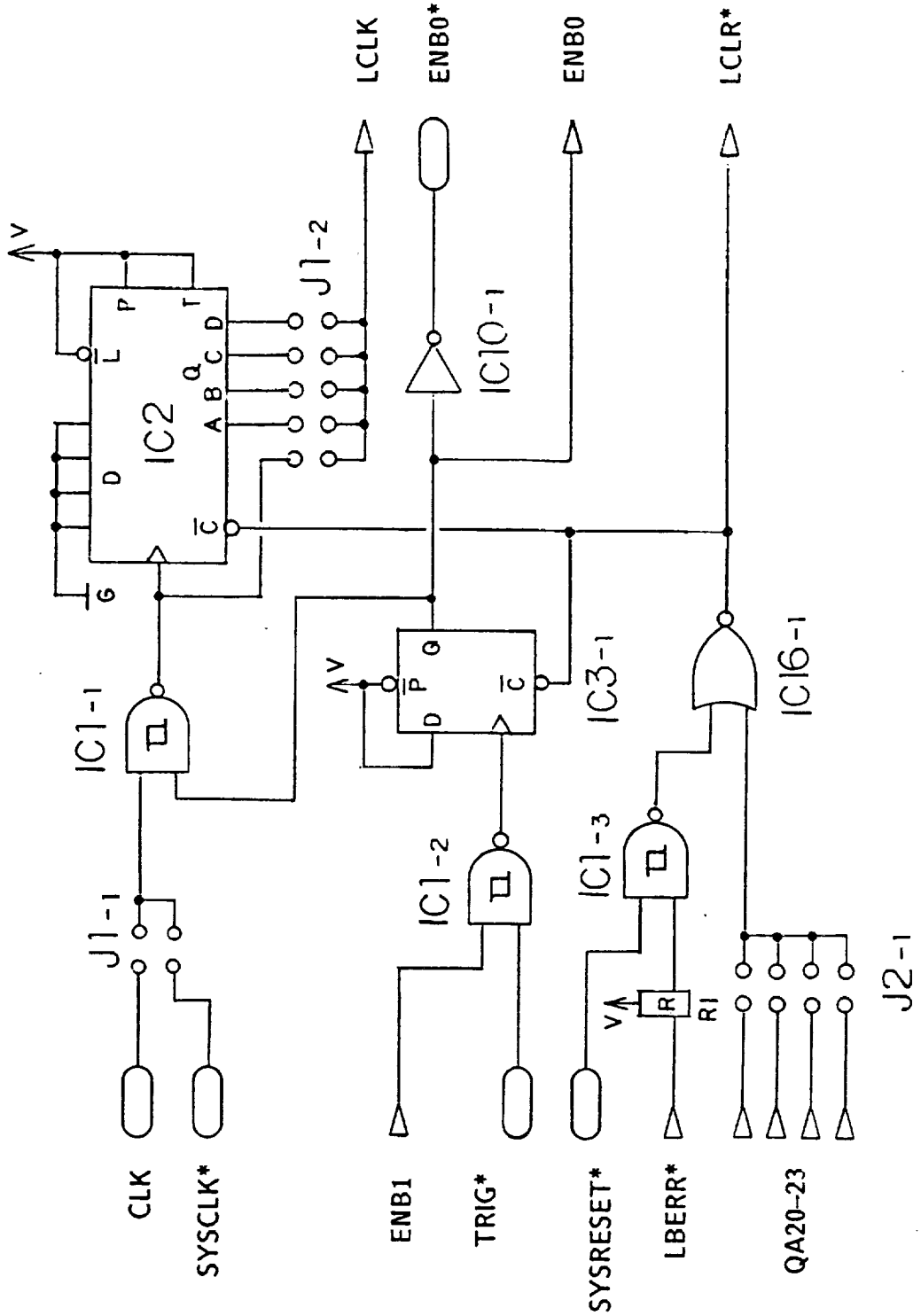
A.2 Address Generator

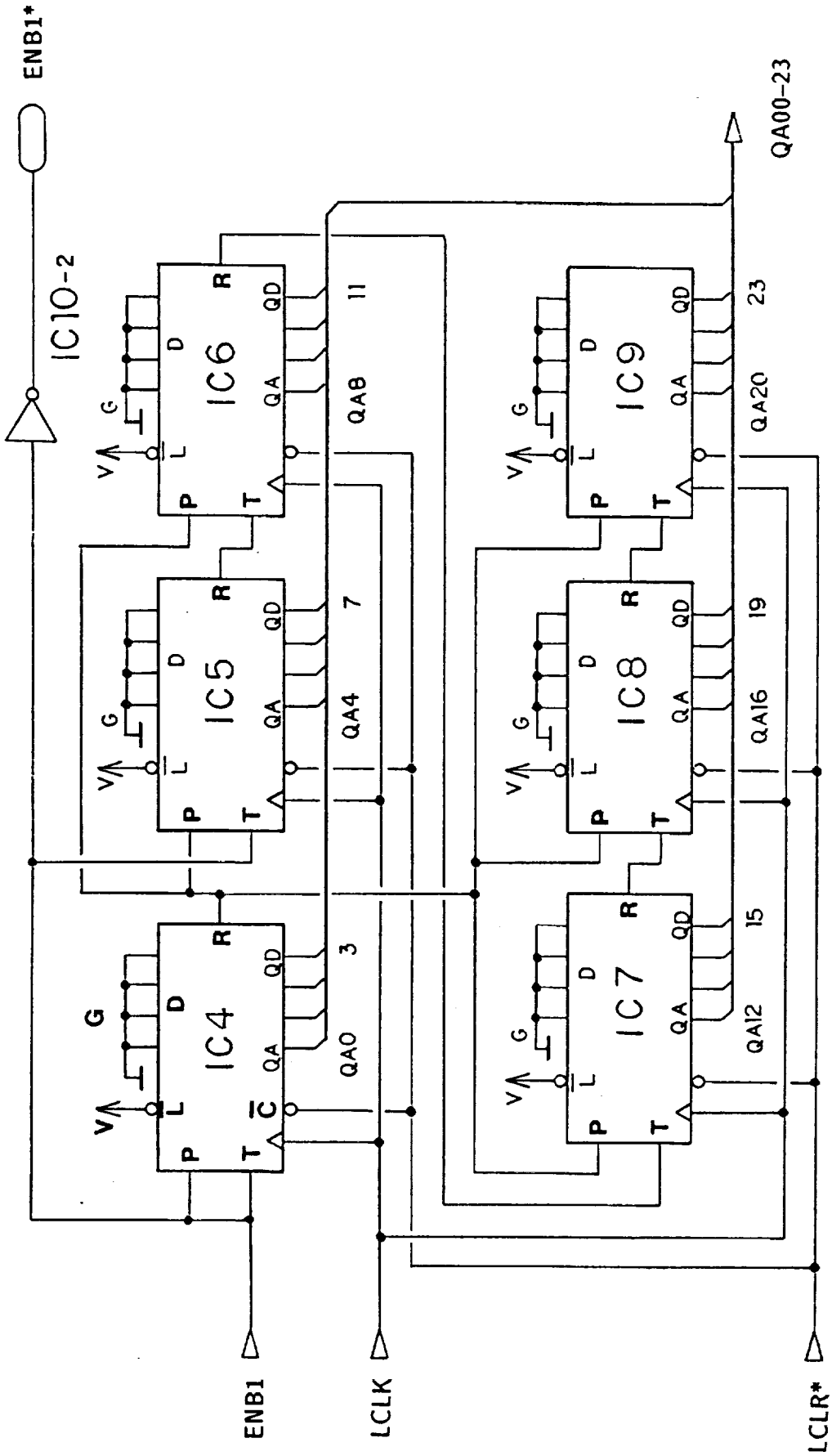
A.3 Address Bus Buffers and Address Modifier Selector

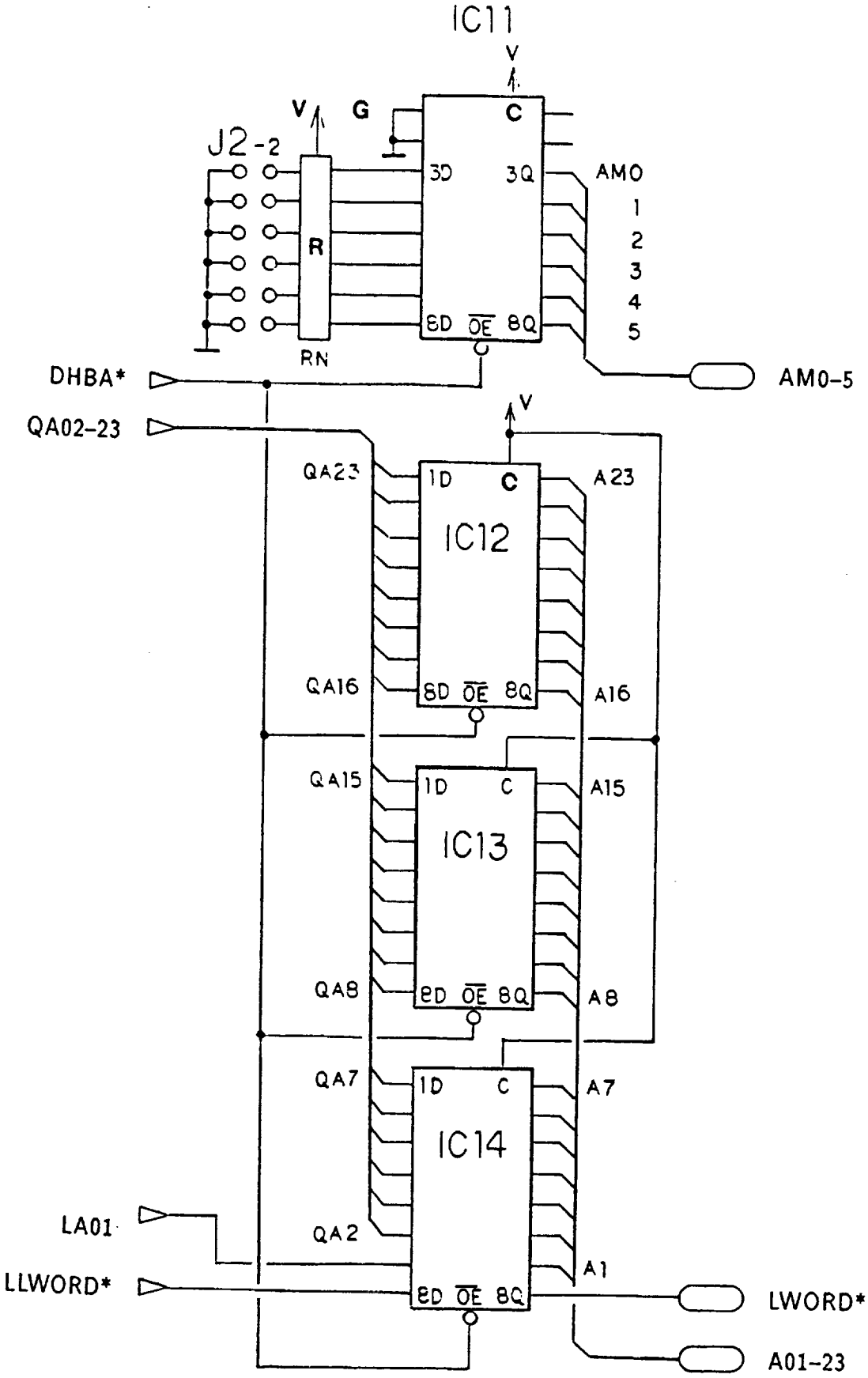
A.4 Data Transfer Control

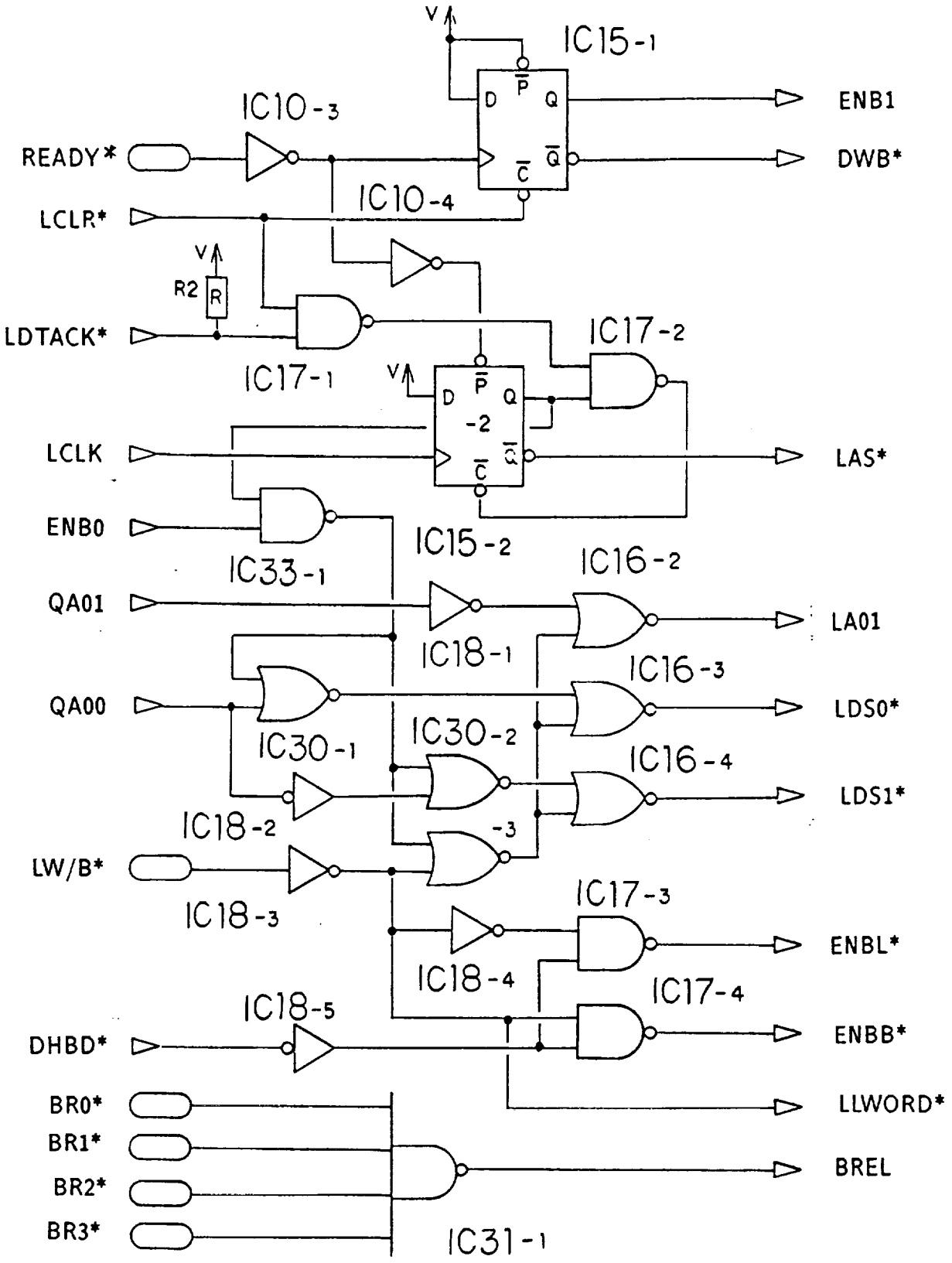
A.5 Input Channel Selector and Data Bus Buffers

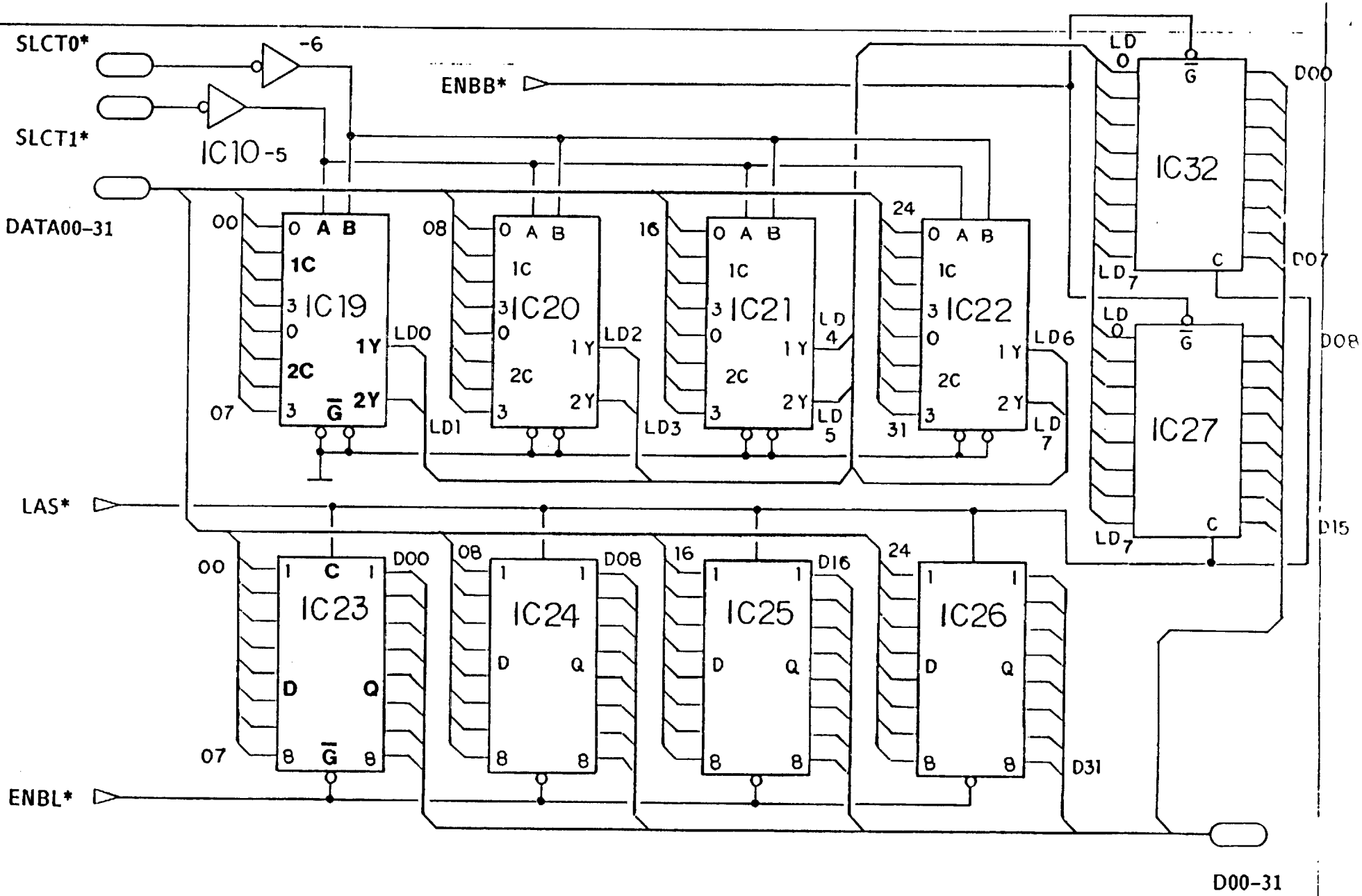
A.6 VMEbus Master Control

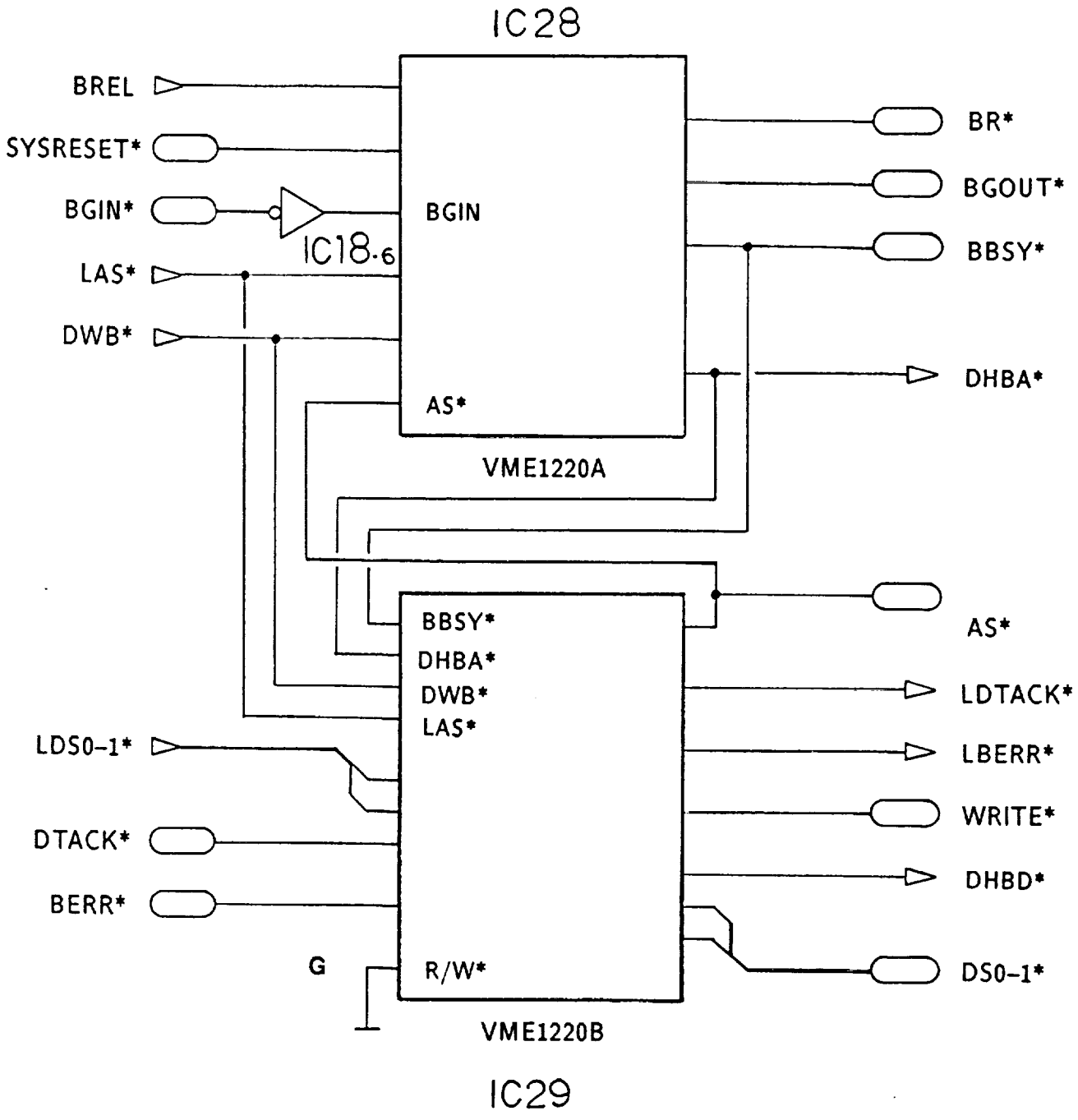

LCLK
ENB0*
ENB0
LCLR*
J1-2
IC10-1
IC2
IC1-1
IC3-1
IC16-1
J1-1
IC1-2
IC1-3
R1
J2-1
CLK
SYSCLK*
ENB1
TRIG*
SYSRESET*
LBERR*
QA20-23


ENB1*
IC10-2
QA00-23
IC6
IC5
IC4
IC9
IC8
IC7
QA8
QA4
QA0
QA20
QA16
QA12
11
7
3
23
19
15
ENB1
LCLK
LCLR*


IC11
V
V
G
C
J2-2
3D
3Q
AM0
1
2
3
4
5
R
8D
OE
8Q
RN
DHBA*
AM0-5
QA02-23
V
QA23
1D
C
A23
IC12
QA16
8D
OE
8Q
A16
QA15
1D
C
A15
IC13
QA8
8D
OE
8Q
A8
QA7
1D
C
A7
IC14
LA01
QA2
A1
LLWORD*
8D
OE
8Q
LWORD*
A01-23


V
IC15-1
D
P
Q
C
Q
ENB1
DWB*
IC10-3
READY*
IC10-4
LCLR*
V
R2
R
LDTACK*
IC17-2
IC17-1
V
D
P
Q
-2
C
Q
LAS*
LCLK
ENB0
IC15-2
IC33-1
IC16-2
QA01
LA01
IC18-1
IC16-3
QA00
LDS0*
IC30-1
IC30-2
IC16-4
LDS1*
IC18-2
-3
LW/B*
IC18-3
IC17-3
ENBL*
IC18-4
IC17-4
IC18-5
DHBD*
ENBB*
LLWORD*
BR0*
BR1*
BR2*
BR3*
BREL
IC31-1


SLCT0*
SLCT1*
DATA00-31
LAS*
ENBL*
ENBB*
IC10-5
-6
IC19
IC20
IC21
IC22
IC23
IC24
IC25
IC26
IC27
IC32
LD0
LD1
LD2
LD3
LD4
LD5
LD6
LD7
D00
D07
D08
D15
D16
D31
D00-31


IC28
BREL
SYSRESET*
BGIN*
IC18-6
LAS*
DWB*
BGIN
AS*
VME1220A
BR*
BGOUT*
BBSY*
DHBA*
BBSY*
DHBA*
DWB*
LAS*
LDS0-1*
DTACK*
BERR*
G
R/W*
VME1220B
AS*
LDTACK*
LBERR*
WRITE*
DHBD*
DS0-1*
IC29

Appendix B Parts List

Table B.1 DAM Parts List (1)

LABEL	Part Number	Pins	DESCRIPTION
IC1	74LS132	14	Quadruple Schmitt NAND gates
IC2	74LS161A	16	Synchronous 4-bit counter
IC3	74AS74	14	Dual D-type F/Fs
IC4	74LS161A	16	Synchronous 4-bit counter
IC5	74LS161A	16	
IC6	74LS161A	16	
IC7	74LS161A	16	
IC8	74LS161A	16	
IC9	74LS161A	16	
IC10	74LS04	14	Hex inverters
IC11	74AS573	20	Octal D-type transparent latches
IC12	74AS573	20	
IC13	74AS573	20	
IC14	74AS573	20	
IC15	74AS74	14	Dual D-type F/Fs
IC16	74AS02	14	Quadruple 2-input NOR gates
IC17	74AS00	14	Quadruple 2-input NAND gates
IC18	74AS04	14	Hex inverters
IC19	74LS153	16	Dual 4-to-1 data selectors

Table B.2 DAM Parts List (2)

LABEL	Part Number	Pins	DESCRIPTION
IC20	74LS153	16	Dual 4-to-1 data selectors
IC21	74LS153	16	
IC22	74LS153	16	
IC23	74AS573	20	Octal D-type transparent latches
IC24	74AS573	20	
IC25	74AS573	20	
IC26	74AS573	20	
IC27	74AS573	20	
IC28	VME1220A	24	VMEbus master controller
IC29	VME1220B	24	(Non-slot 1, P-45)
IC30	74AS02	14	Quadruple 2-input NOR gates
IC31	74LS20	14	Dual 4-input NAND gates
IC32	74AS573	20	Octal D-type transparent latches
IC33	74AS00	14	Quadruple 2-input NAND gates

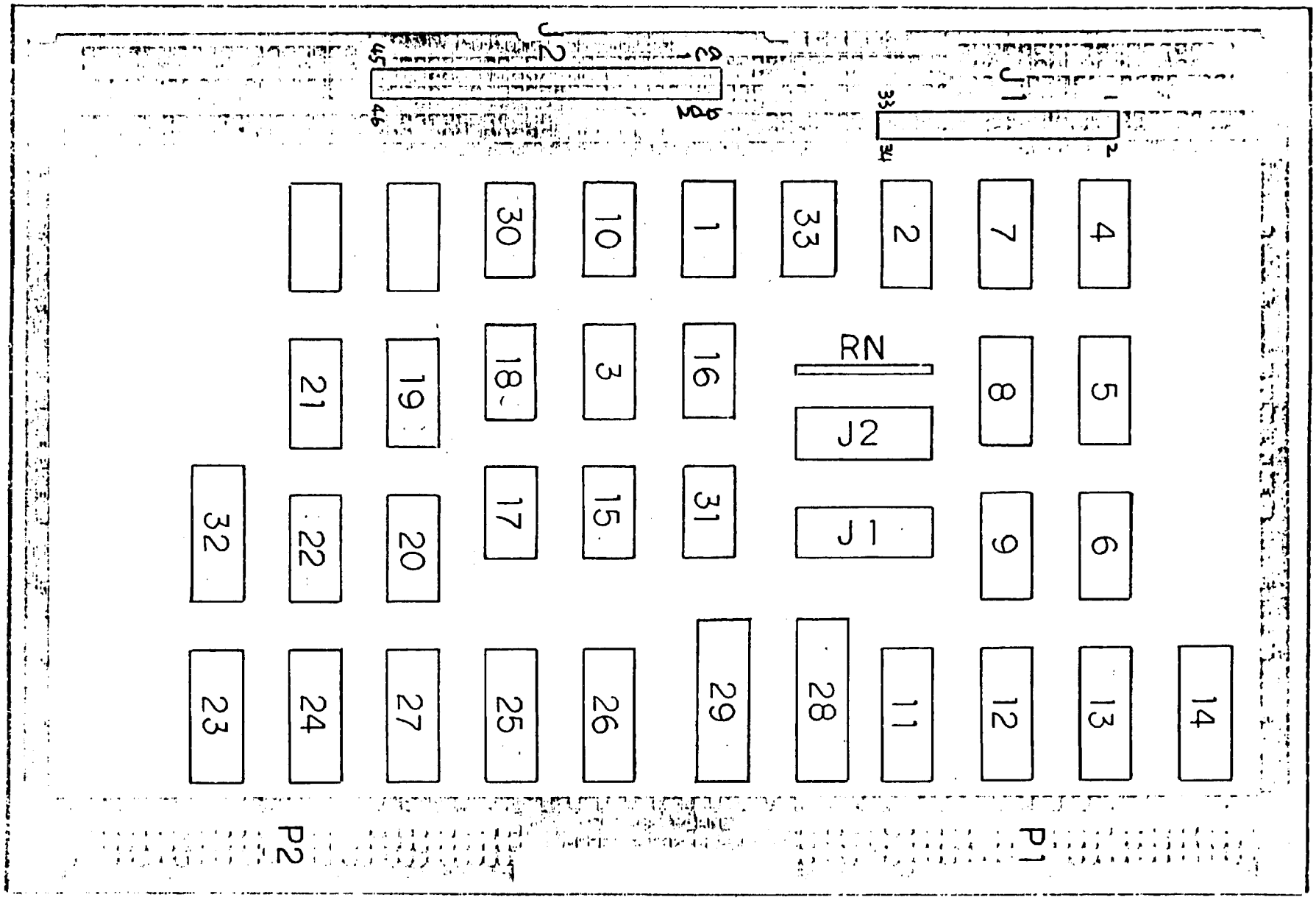
Appendix C DAM Board Layout

C.1 Component Side Layout

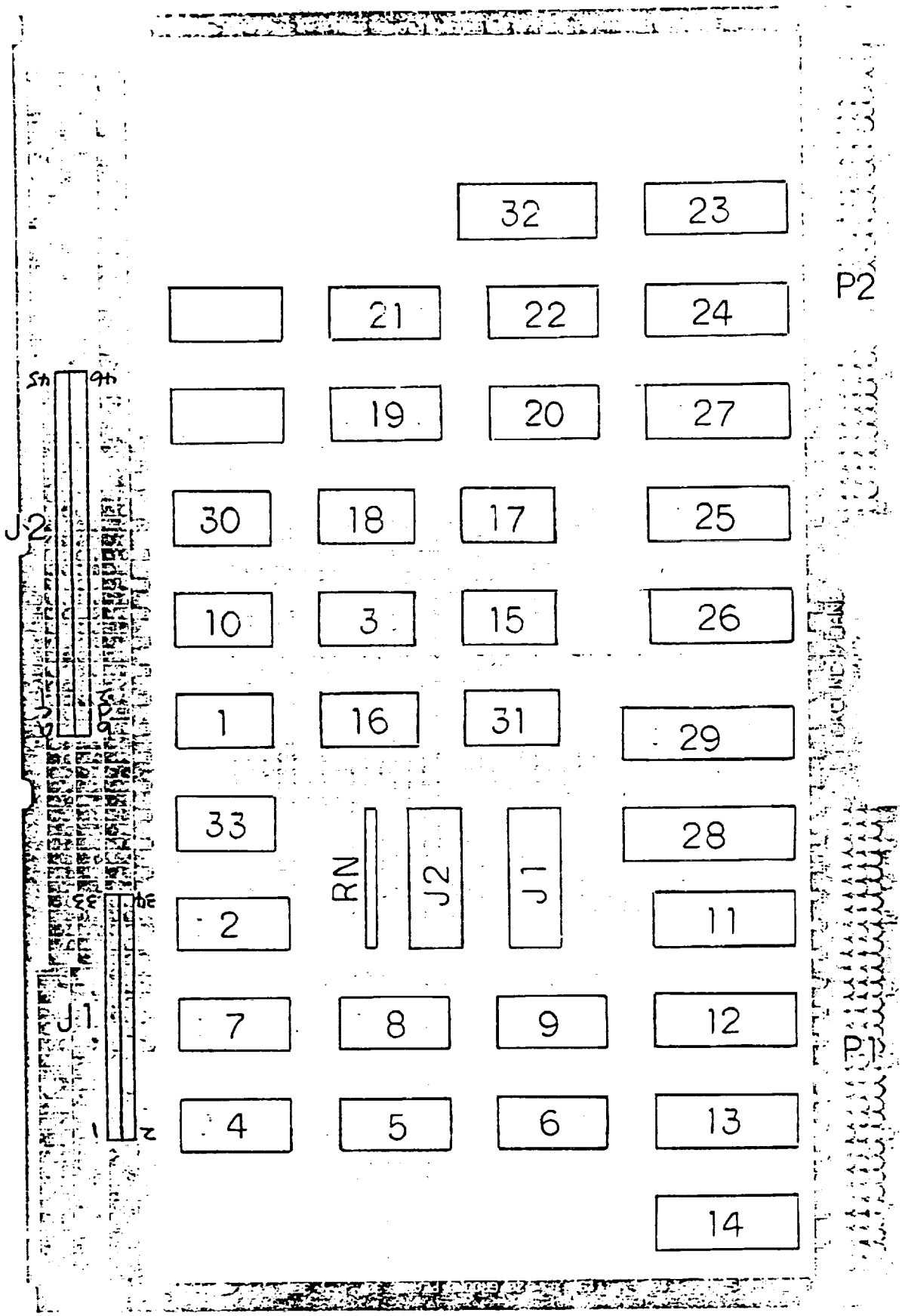
C.2 Wiring Side Layout

E160-6U-3 EUROCARD
COMPONENT SIDE LAYOUT PAPER

LA52P1



ELECTRONIC COMPANY
12460 GLADSTONE AVE.
SYLMAR, CA 91342



Appendix D Copies of Data Sheets

D.1 VME 1220 Non-Slot 1 VMEbus Master Controller

VME 1210/1220

June 1990

Slot 1 and Non-Slot 1 VMEbus Master Controllers

Distinctive Features

- **VME 1210 provides two device chip set for slot 1 master bus controller and single level arbiter**
- **VME 1220 provides two device chip set for non-slot 1 master bus controller**
- **Integrates 48ma and 64ma VMEbus signals:AS*,DS0*,DS1*,WRITE*,BR*,BBSY***
- **Integrates input hysteresis buffers**
- **Supports Release When Done (RWD) and Release On Request (ROR) protocols**
- **Supports address pipelining, block transfers, and early BBSY* release**
- **Available in Commercial, Industrial and Military temperature ranges**

Programmable Version Available

If the VME 1210/1220 does not match the requirements of the design, a programmable version is available (the PLX 464) which allows the user to customize all inputs, outputs and logic. Programming is performed using industry standard tools such as ABEL™ and CUPL™ software and commonly available PLD programming hardware. Contact PLX for a data sheet on the PLX 464 and other PLX products.

Applications

- VMEbus masters residing in slot 1 boards (VME 1210)
- VMEbus masters residing in non-slot 1 boards (VME 1220)

General Description

The VME 1210: The VME 1210 is comprised of the VME 1210A and the VME 1210B for slot 1 applications. The devices are CMOS and packaged in 24 pin 300 mil wide DIPs or 28 pin J-type LCCs. The VME 1210A provides bus requesting, local arbitration, and single level system arbitration. The VME 1210B functions as the VMEbus controller. The requester initiates a VMEbus request from the local master's bus request for a data or interrupt cycle. The bus controller controls the bus after initiation of a bus cycle and relinquishes the bus at the end of the bus cycle. The bus controller supervises the handshaking between the local master CPU and the slave modules.

The VME 1220: The VME 1220 is comprised of the VME 1220A and the VME 1220B for non-slot 1 applications. The devices are CMOS and packaged in 24 pin 300 mil wide DIPs or 28 pin J-type LCCs. The VME 1220A provides bus requesting and local arbitration. The VME 1220B functions as the VMEbus controller. The requester initiates a VMEbus request from the local master's bus request for a data or interrupt cycle. The bus controller controls the bus after initiation of a bus cycle and relinquishes the bus at the end of the bus cycle. The bus controller supervises the handshaking between the local master CPU and the slave modules.

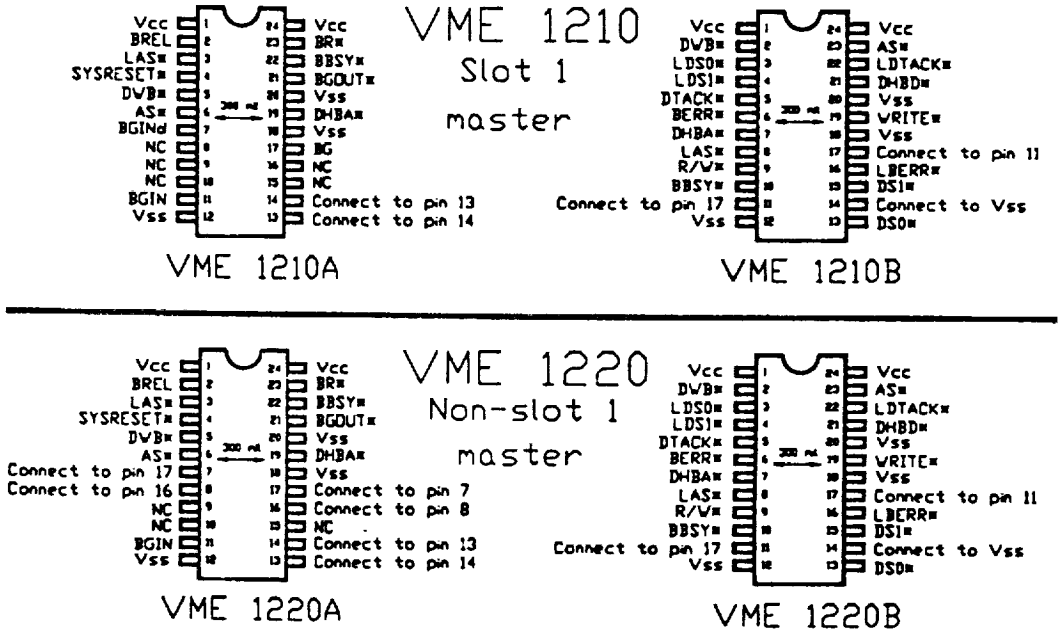


Figure 1. Pinout of VME 1210/1220 (DIPs)

Patent Pending
ABEL is a trademark of Data I/O Corp.
CUPL is a trademark of Logical Devices, Inc.

in Description

VME 1220A

Pin # LCC	Pin # DIP	Signal	Type	Function
3	2	BREL	I	Active high; Bus release signal indicating BBSY* can be released.
4	3	LAS*	I	Active low; Address strobe from local master.
5	4	SYSRESET*	I	Active low; VMEbus System Reset.
6	5	DWB*	I	Active low; Device wants bus, local master requests control of bus.
7	6	AS*	I	Active low; VMEbus Address Strobe.
9	7	-	I	Connect to pin 17 (DIP) or pin 20 (LCC).
10	8	-	I	Connect to pin 16 (DIP) or pin 19 (LCC).
11	9	NC	I	No Connect.
12	10	NC	I	No Connect.
13	11	BGIN	I	Active high; Inverted VMEbus Bus Grant In signal, BGIN*.
14,21, 24	12,18, 20	Vss		Chip Ground.
16	13	-	O	Connect to Pin 14 (DIP) or Pin 17 (LCC).
17	14	-	I	Connect to Pin 13 (DIP) or Pin 16 (LCC).
18	15	NC	O	No Connect.
19	16	-	O	Connect to pin 8 (DIP) or pin 10 (LCC).
20	17	-	O	Connect to pin 7 (DIP) or pin 9 (LCC).
23	19	DHBA*	O	Active low; Device has bus address, address buffer enable.
25	21	BGOUT*	O	Active low; VMEbus Bus Grant Out signal.
26	22	BBSY*	I/O	Active low, 48 mA open collector; VMEbus Bus Busy signal.
27	23	BR*	O	Active low, 48 mA open collector; VMEbus Bus Request signal.
2,28	1,24	Vcc		+5 V Chip Power
1,8, 15,22	-	NC	-	No Connect.

Pin Description

VME 1210B and VME1220B

Pin # LCC	Pin # DIP	Signal	Type	Function
3	2	DWB*	I	Active low; Device wants bus, local master wants control of VMEbus.
4	3	LDS0*	I	Active low; Lower data strobe from local master.
5	4	LDS1*	I	Active low; Upper data strobe from local master.
6	5	DTACK*	I	Active low; VMEbus Data Transfer Acknowledge, data is valid during a read cycle or data has been accepted from the bus during a write cycle.
7	6	BERR*	I	Active low; VMEbus Error signal.
9	7	DHBA*	I	Active low; Device has bus address, address buffer enable.
10	8	LAS*	I	Active low; Address strobe from local master.
11	9	R/W*	I	Active high/low; Read or write cycle from local master.
12	10	BBSY*	I	Active low; VMEbus Busy, local master controls bus.
13	11	-	I	Connect to pin 17 (DIP) or pin 20 (LCC).
14,21, 24	12,18, 20	Vss		Chip Ground.
16	13	DS0*	O	Active low; 64ma VMEbus lower Data Strobe signal, indicates valid data on bus.
17	14	-	I	Connect to Vss.
18	15	DS1*	O	Active low; 64ma VMEbus upper Data Strobe signal, indicates valid data on bus.
19	16	LBERR*	O	Active low; Open collector signal, bus error to local master.
20	17	-	O	Connect to pin 11 (DIP) or pin 13 (LCC).
23	19	WRITE*	O	Active low; 48ma VMEbus Write signal, indicates bus read or write cycle.
25	21	DHBD*	O	Active low; Device has bus data, data buffer enable.
26	22	LDTACK*	O	Active low; Open collector signal, data acknowledge to local master.
27	23	AS*	O	Active low; 64mA VMEbus Address Strobe signal, indicates valid address on bus.
2,28	1,24	Vcc		+5 V Chip Power
1,8, 15,22	-	NC	-	No Connect.

VME 1210/1220 Timing Waveforms

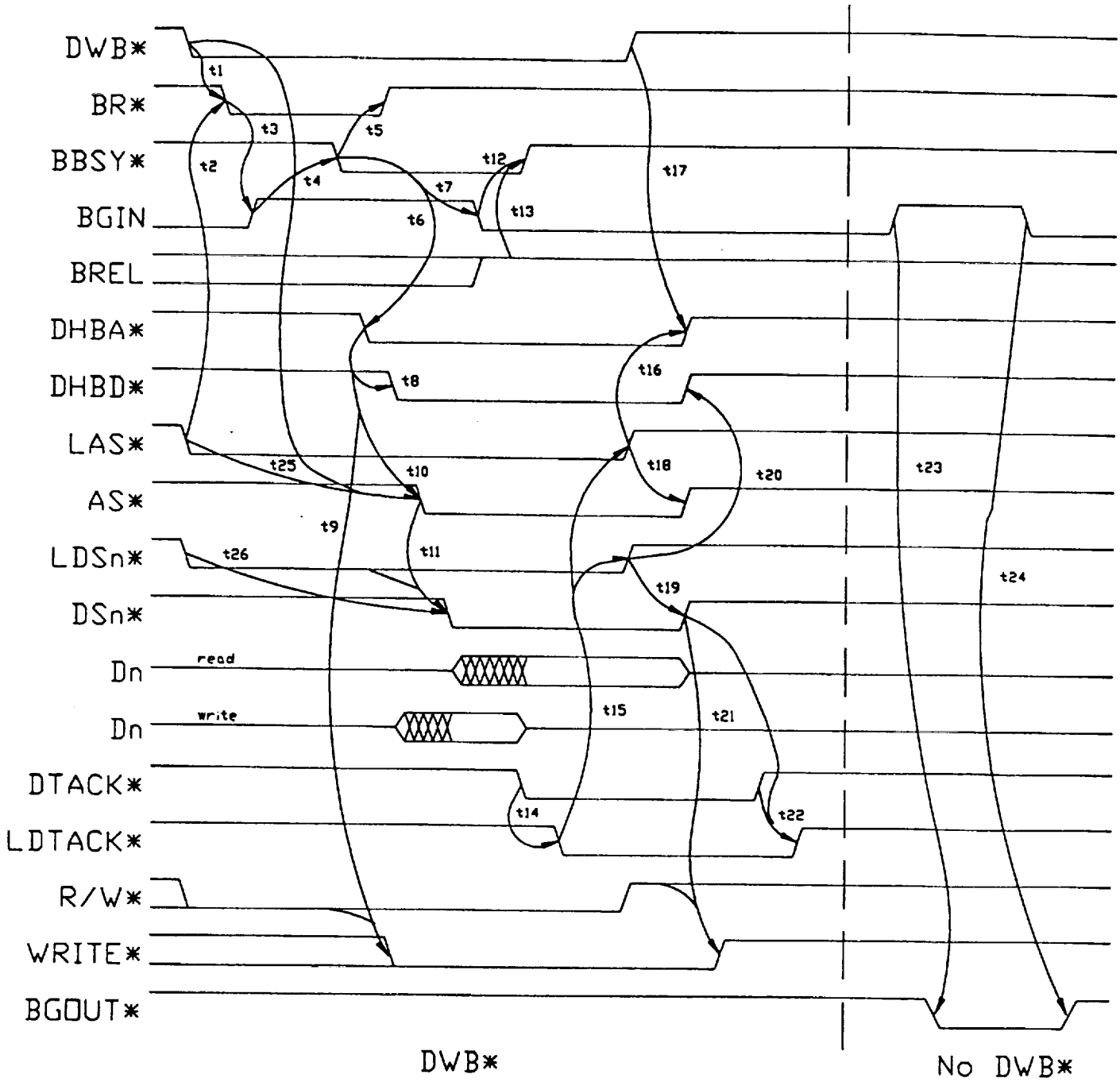


Figure 5. Timing Diagram

6-8

PRECEDING PAGE BLANK NOT FILMED

Timing Specifications

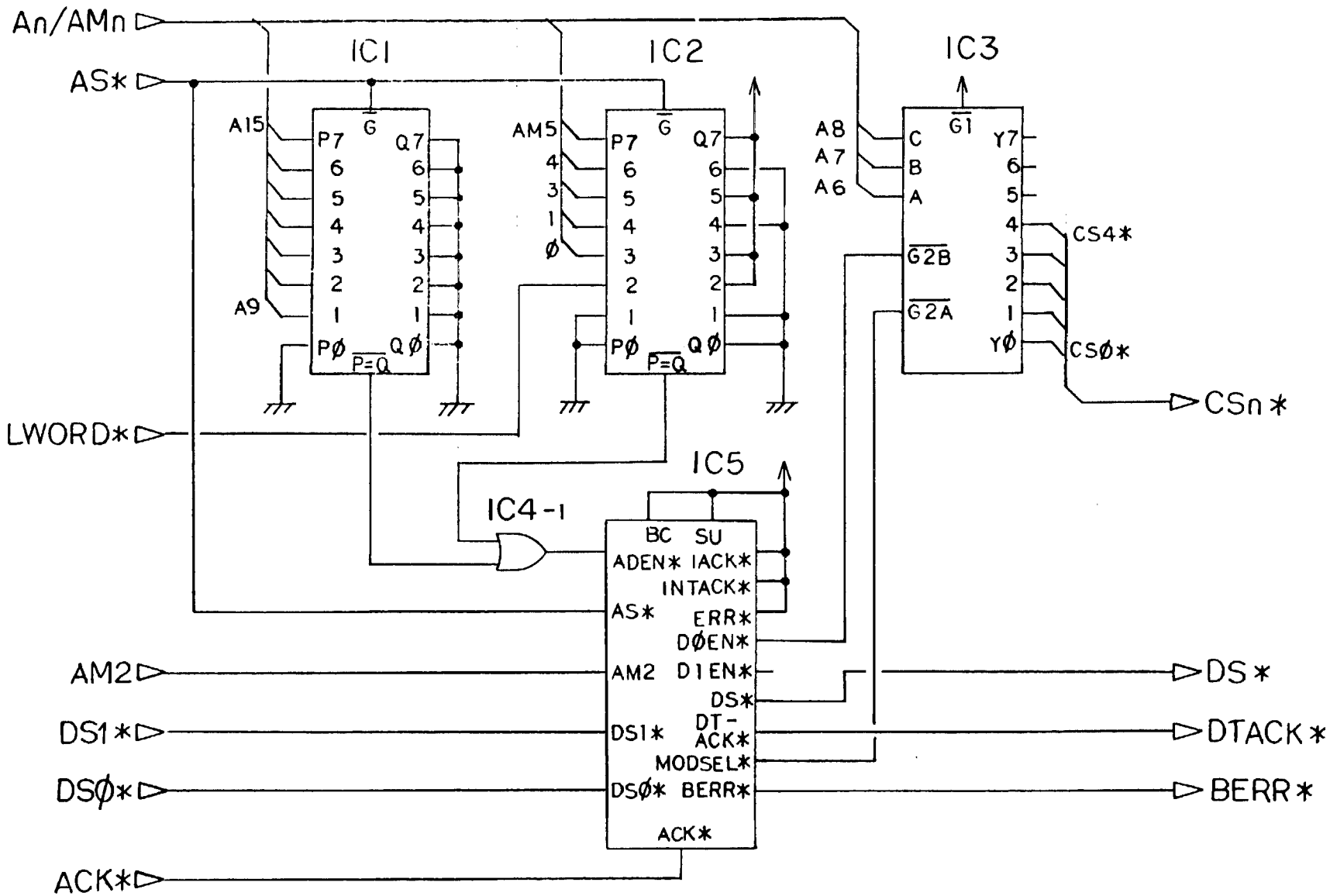
Timing Parameters	Signals	Max. Time(ns) unless otherwise specified		Description
		C-45	M-65	
t1	DWB* to BR* asserted	90	130	If DWB* is asserted after LAS*
t2	LAS* to BR* asserted	90	130	If LAS* is asserted after DWB*
t3	BR* to BG asserted	0	0	VME 1210 only when internal BR* generated (BG connected to BGIN)
		45	65	VME 1210 only when external BR* received (BG connected to BGIN)
		System arbiter time	System arbiter Time	VME 1220 only
t4	BGIN to BBSY* asserted	125	185	VME 1210 only, includes delay line: 55ns for M-65, 45ns for M-55, 35ns for C-45, 40ns for C-35, 60ns for C-25 part
		135	195	VME 1220 only
t5	BBSY* to BR* negated	45	65	
t6	BBSY* to DHBA* asserted	45	65	
t7	BBSY* to BGIN negated	45 max	65	VME 1210 only
		35 min	55 min	
		System arbiter time	System arbiter time	VME 1220 only
t8	DHBA* to DHBD* asserted	45	65	
t9	DHBA* to WRITE* asserted	45	65	Conditional upon R/W* value
t10	DHBA* to AS* asserted	90 70 (min.)	130	Ensures 35ns minimum address to AS* and data to DSn* set up times
t11	AS* to DSn* asserted	45	65	
t12	BGIN to BBSY* negated	80 max	120 max	VME 1210 only;
		70 min	110 min	VME 1210 only; t7min + t12min ≥ 90 ns min. BBSY* assertion
		135 max	195 max	VME 1220 only
		105 min	165 min	VME 1220 only. (see note below)
t13	BREL to BBSY* negated	45	65	Valid only when BREL is asserted after BGIN is negated
t14	DTACK* to LDTACK* asserted	45	65	
t15	LDTACK* to LAS*/LDSn* negated	@ Local master	@ Local master	Local master's time to negate strobes
t16	LAS* to DHBA* negated	45	65	If DWB* already negated
t17	DWB* to DHBA* negated	45	65	If LAS* already negated
t18	LAS* to AS* negated	50	72	
t19	LDSn* to DSn* negated	50	72	
t20	LDSn* to DSn* negated	50	72	
t21	DSn* to WRITE* negated	45	65	Ensures 10ns hold time
t22	DSn*/DTACK* to LDTACK* negated	45	65	Earliest negation of DSn* or DTACK* causes LDTACK* to be negated.
t23	BGIN to BGOUT* asserted	90	130	VME 1220 only
		25+d,35+d,45+d	55+d,65+d	VME 1210 only
t24	BGIN to BGOUT* negated	45	65	
t25	Latest of LAS*/DWB* to AS* asserted	135	195	Assertion time when already have bus (BBSY* asserted).
t26	Latest of DHBD*/LDS* to DS* asserted	45	65	Assertion time when already have bus (BBSY* asserted)

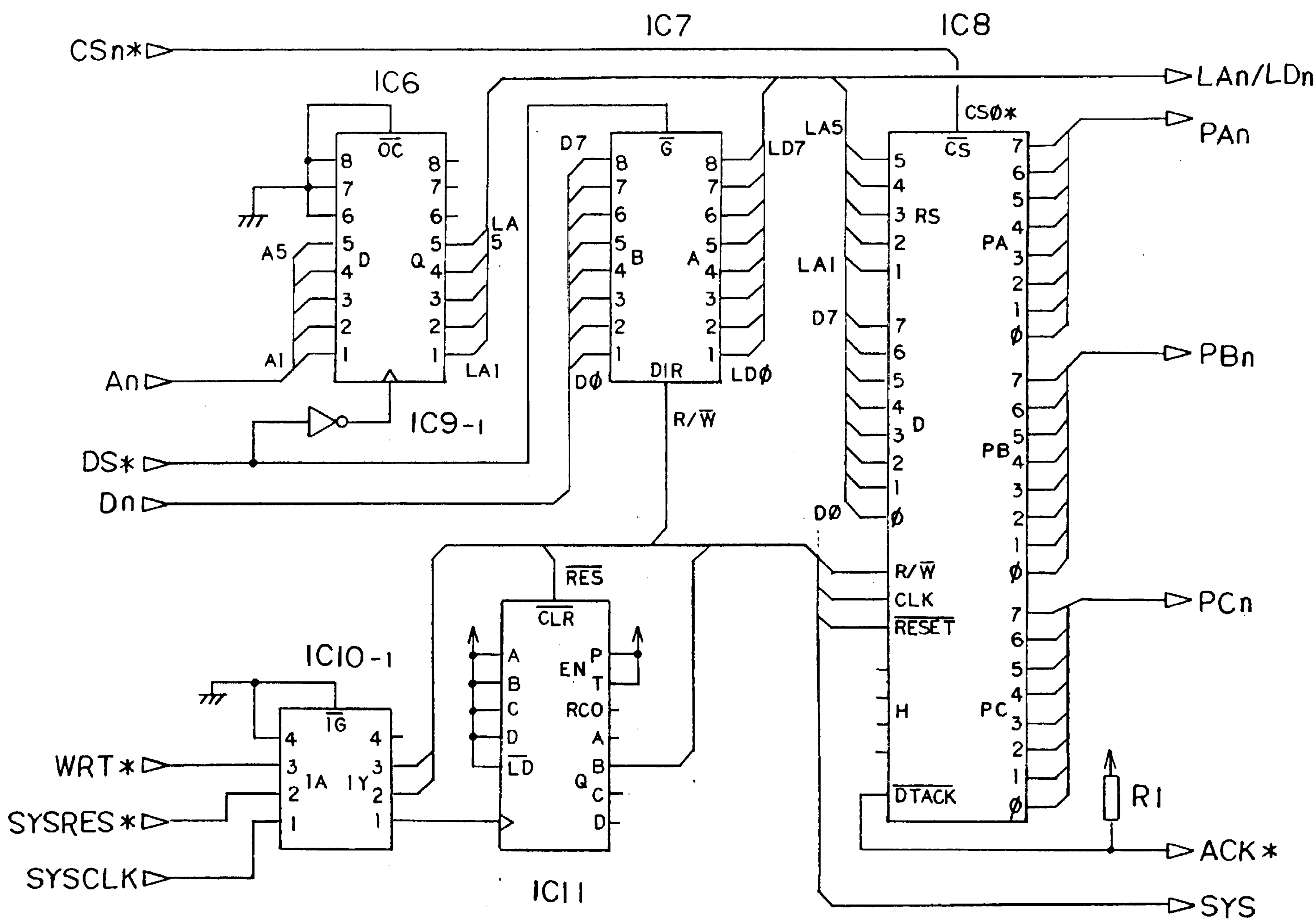
Note:
BBSY* is guaranteed to be asserted for a minimum of 90 ns in the VME 1210A devices and the C-45 device of the VME 1220A, even if BGIN is negated immediately after BBSY* is asserted. For the C-35 and C-25 VME 1220A devices, the sum of the system arbiter "BBSY* asserted to BGIN* negated" time and the t12 minimum time on the VME 1220A must be greater than 90 ns. Generally, this time will be taken up completely by the system arbiter time, however, if not, a delay line can be connected between pins 8 and 16 (DIP) or pins 10 and 19 (LCC) on the VME 1220A device to guarantee the 90 ns minumum. For example, if the system arbiter "BBSY* asserted to BGIN* negated" time was 35ns (min), no delay line would be needed for the C-35 VME 1220A device, since 35 + 75 > 90. However, a 10 ns delay line would be required for the C-25 VME 1220A.

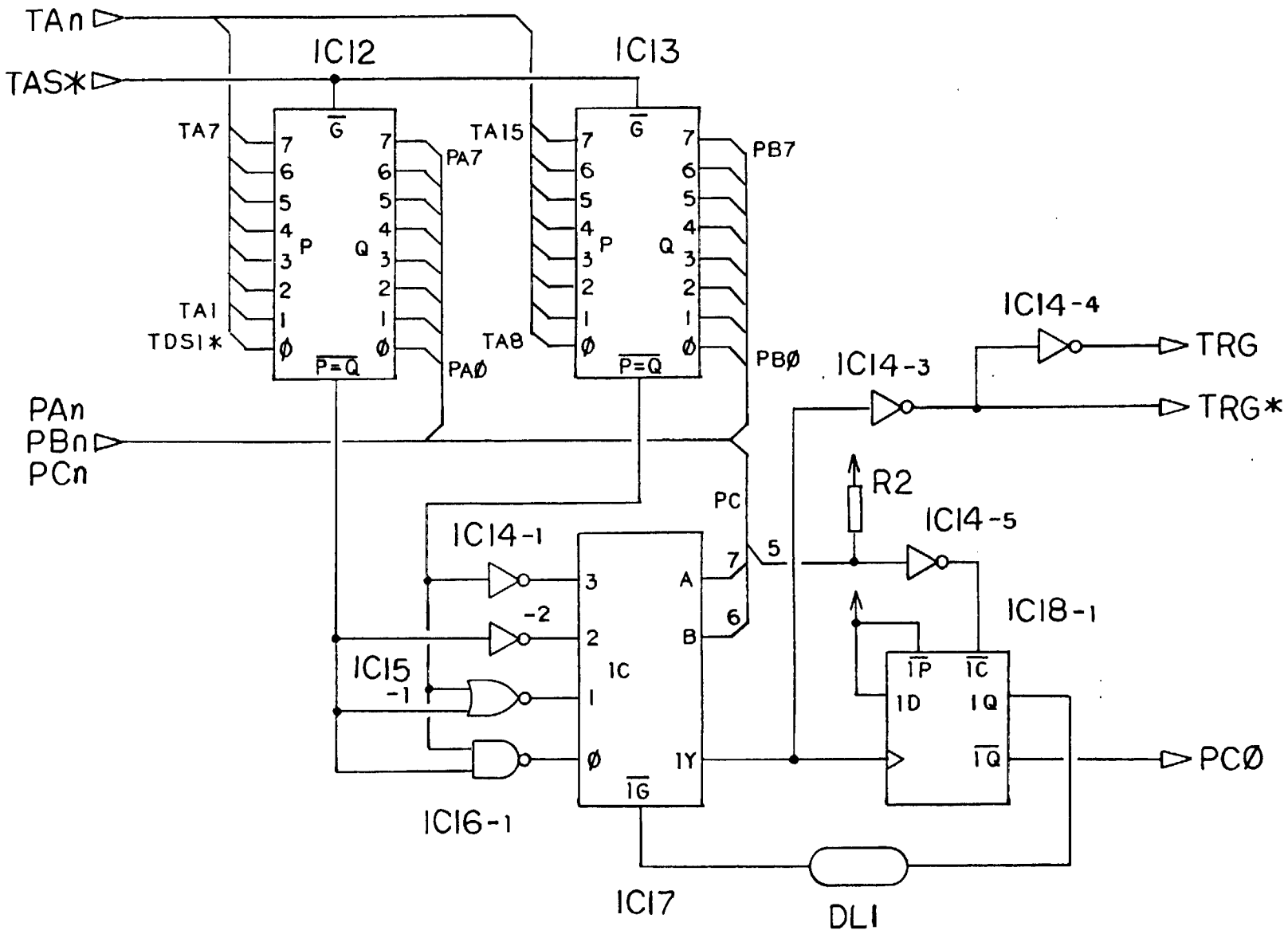
APPENDIX B

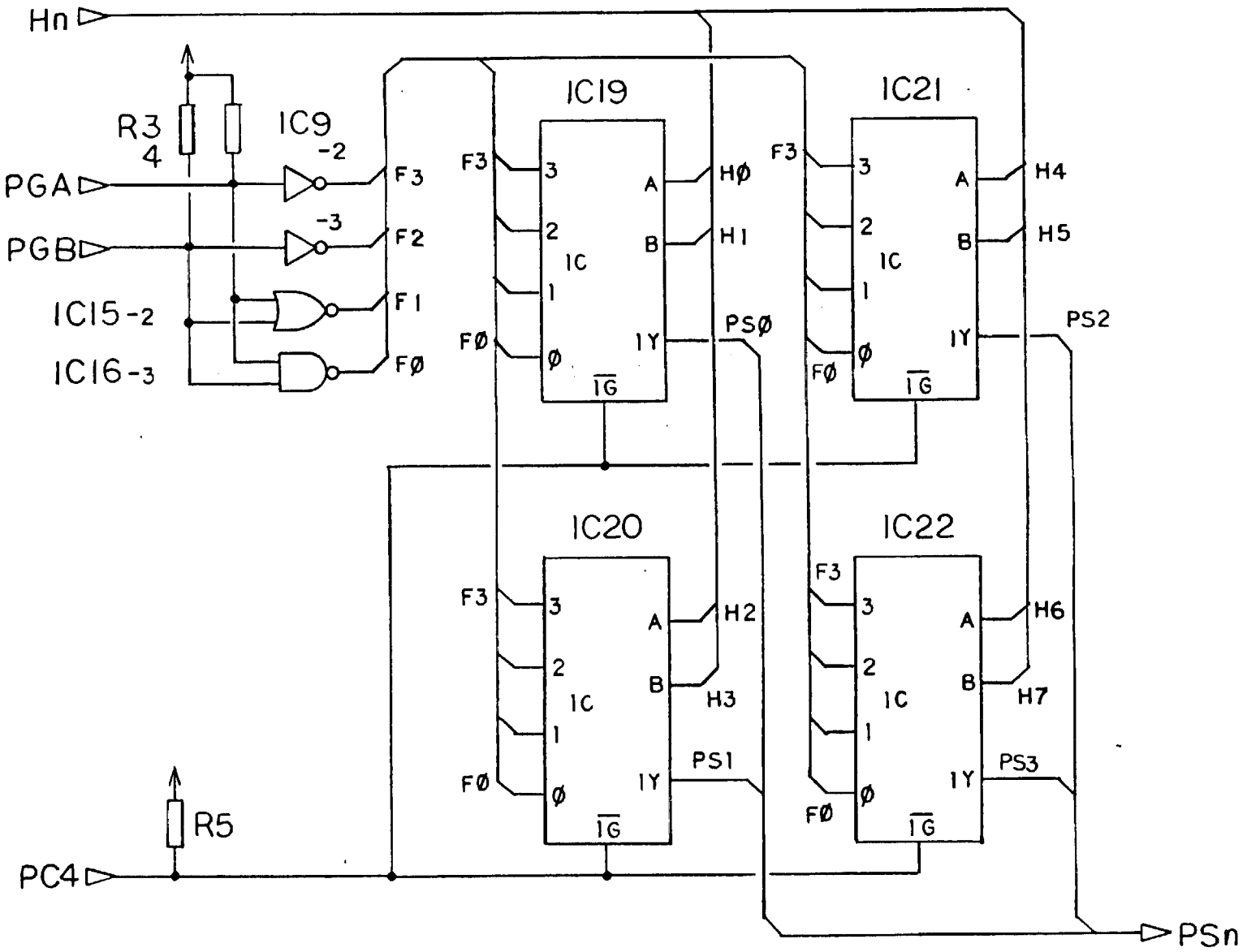
FAULT INJECTION MODULE SCHEMATIC DIAGRAMS

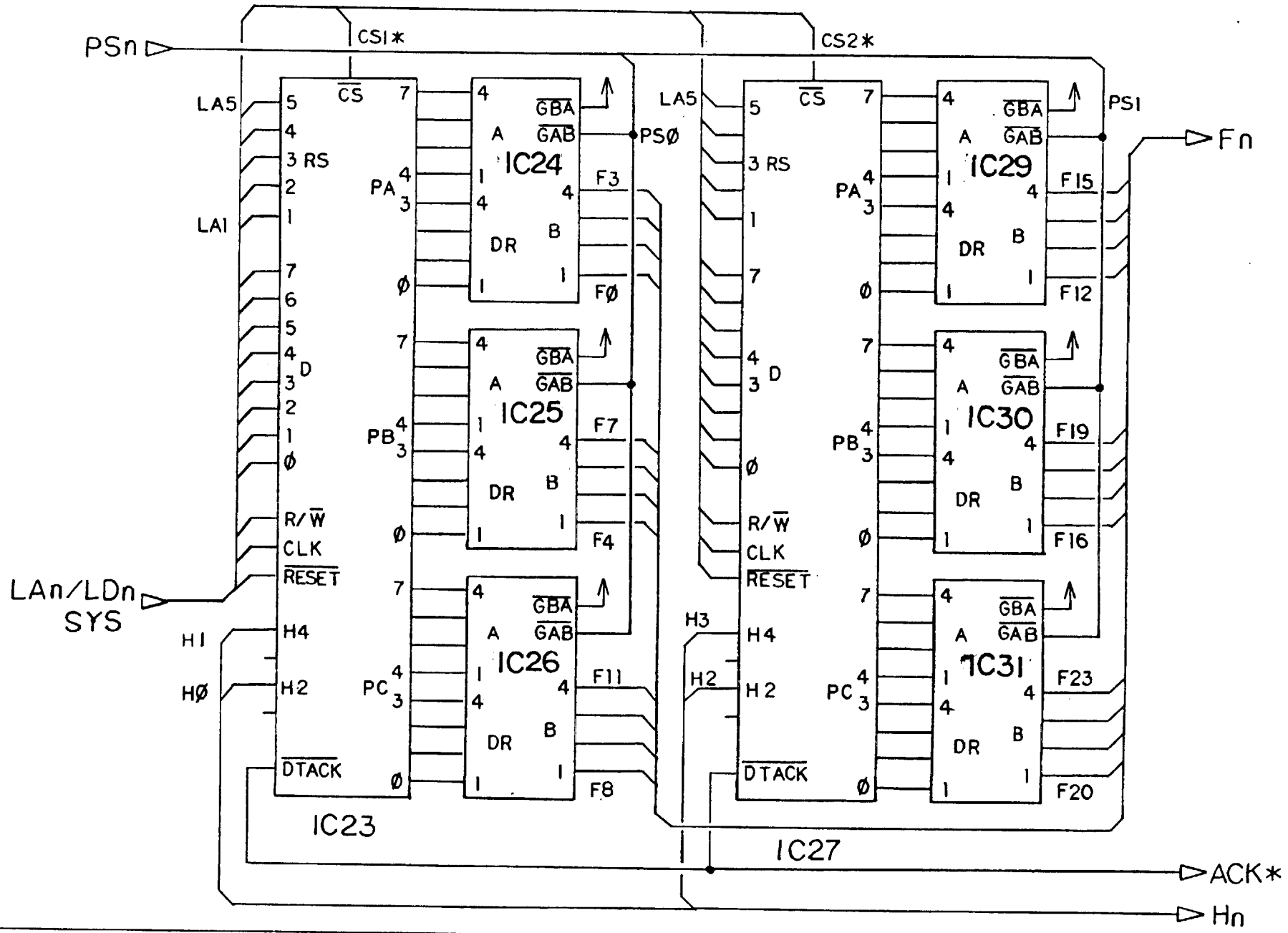
Ver. 1.0

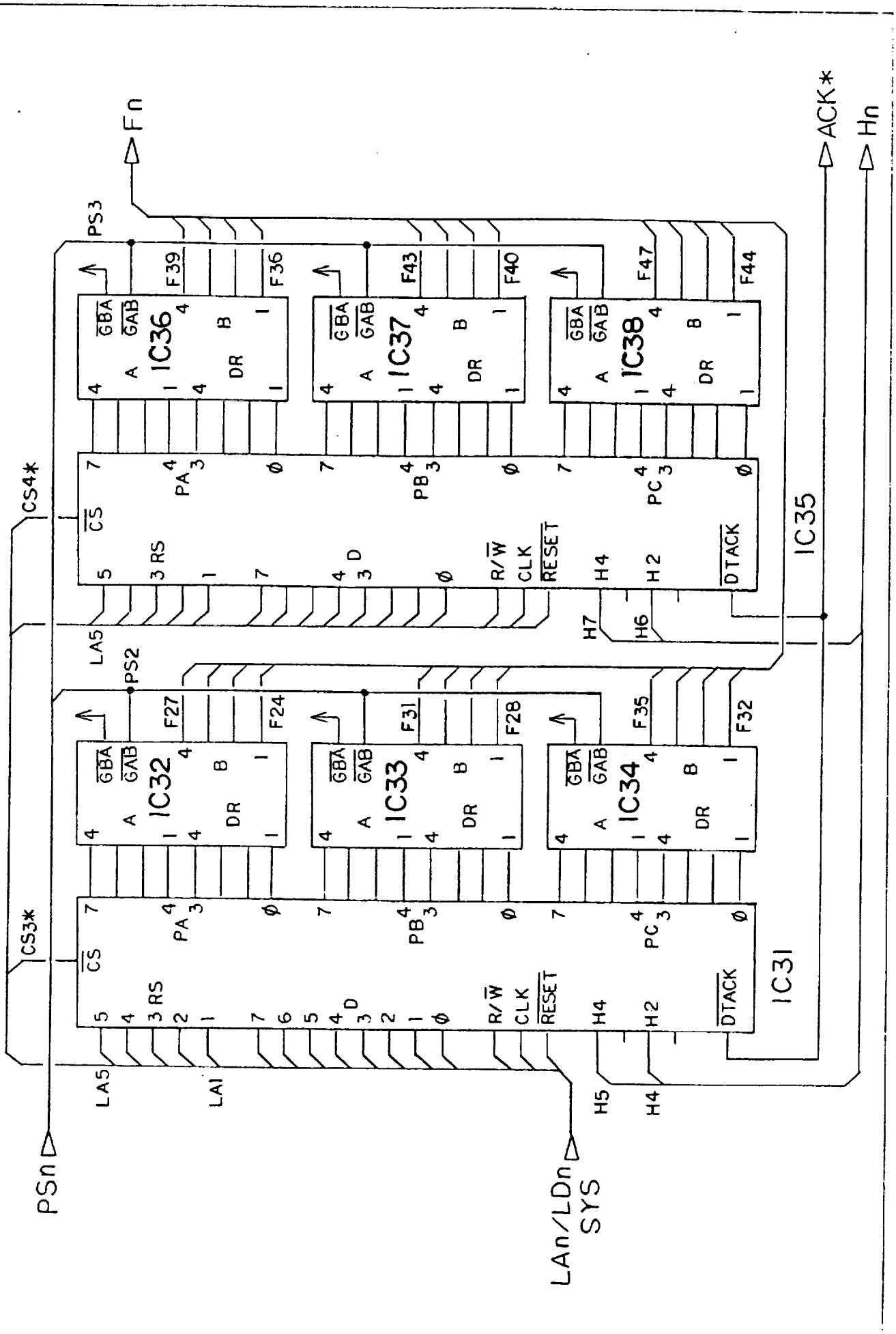

An/AMn
AS*
LWORD*
AM2
DS1*
DSØ*
ACK*
IC1
IC2
IC3
IC4-1
IC5
A15
A9
AM5
A8
A7
A6
P7
Q7
G
P=Q
PØ
QØ
C
B
A
G1
G2B
G2A
Y7
YØ
CS4*
CSØ*
CSn*
BC
SU
ADEN*
IACK*
INTACK*
ERR*
DØEN*
D1EN*
DS*
DT-ACK*
MODSEL*
BERR*
DTACK*
BERR*


IC6
IC7
IC8
IC9-1
IC10-1
IC11
CSn*
An
DS*
Dn
WRT*
SYSRES*
SYSCLK
LAn/LDn
PAn
PBn
PCn
ACK*
SYS
R1
OC
G
CS
CS0*
RS
PA
PB
PC
D
Q
A
B
A5
A1
LA5
LA1
LD7
LD0
D7
D0
DIR
R/W
CLK
RESET
H
DTACK
RES
CLR
EN
P
T
RCO
LD
1G
1A
1Y




Hn
R3
4
IC9
-2
-3
PGA
PGB
IC15-2
IC16-3
F3
F2
F1
FØ
IC19
IC20
IC21
IC22
IC
1G
HØ
H1
H2
H3
H4
H5
H6
H7
PSØ
PS1
PS2
PS3
R5
PC4
PSn


PSn
CS1*
CS2*
LA5
LA1
RS
D
R/W
CLK
RESET
H4
H2
DTACK
CS
IC23
IC24
IC25
IC26
IC27
IC29
IC30
IC31
GBA
GAB
A
B
DR
PA
PB
PC
PSØ
PS1
F3
FØ
F7
F4
F11
F8
F15
F12
F19
F16
F23
F20
LAn/LDn
SYS
H1
HØ
H3
Fn
ACK*
Hn


Fn
ACK*
Hn
PS3
PS2
CS4*
CS3*
IC36
IC37
IC38
IC32
IC33
IC34
IC35
IC31
F39
F36
F43
F40
F47
F44
F27
F24
F31
F28
F35
F32
GBA
GAB
A
B
DR
PA
PB
PC
CS
RS
D
R/W
CLK
RESET
DTACK
H4
H2
H7
H6
H5
LA5
LA1
PSn
LAn/LDn
SYS

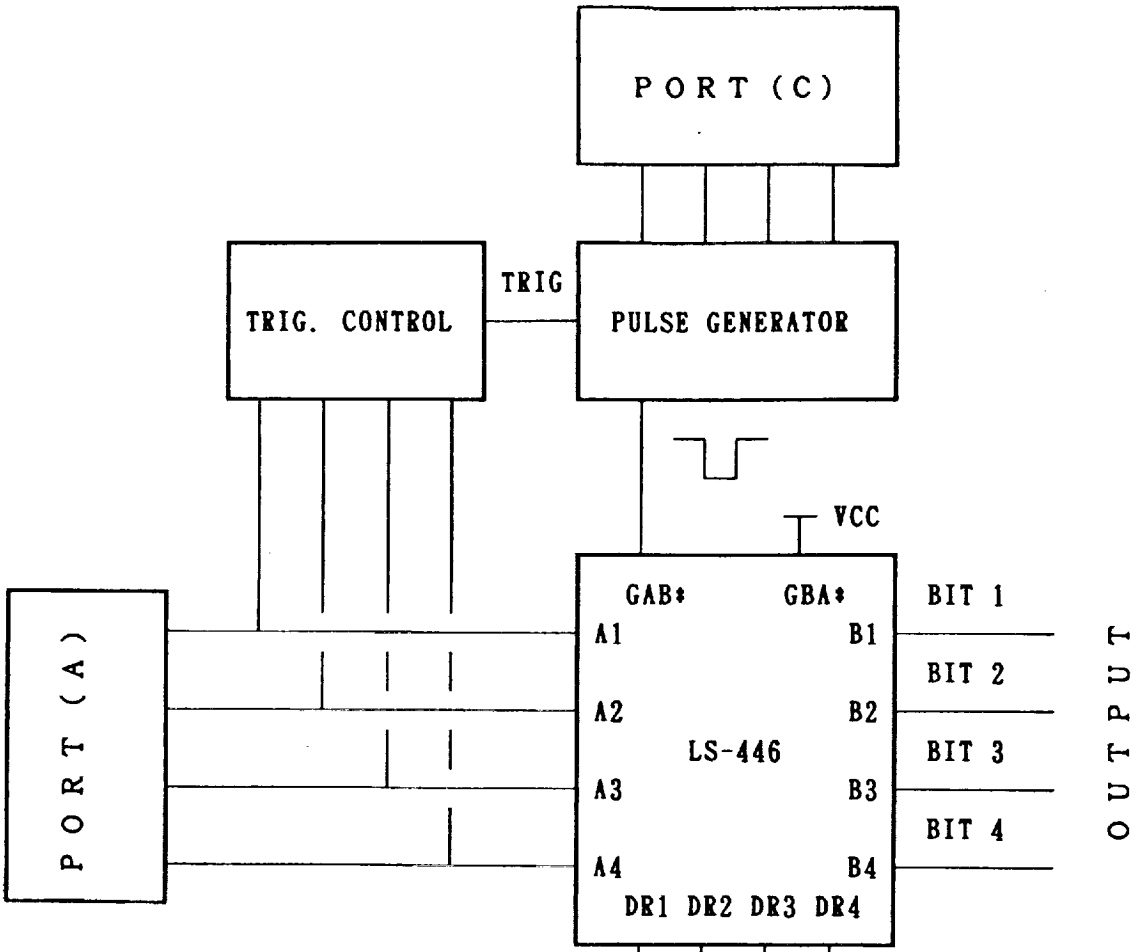
Fault Injection Module
Parts List (1)

Ref No.	Part Number	Size	Description
IC1	SN74ALS520	20	8-bit Identity Comparator
IC2	SN74ALS520	20	8-bit Identity Comparator
IC3	SN74ALS138	16	3 to 8 Decoder
IC4-1	SN74ALS32	14	Quad 2-Input OR Gates (1/4)
IC5	VME 2000	24[1]	Slave Module Interface Device
IC6	SN74F374	20	Octal D-Type Flip-Flops
IC7	SN74LS645-1	20	Octal Bus Transceivers
IC8	MC68230 P8	48	Parallel Interface/Timer (PIT-0)
IC9-1	SN74ALS04B	14	Hex Inverters (1/6)
IC10-1	SN74LS244	20	Octal Buffers (1/2)
IC11	SN74ALS161B	16	4-bit Binary Counter
R1		8[2]	R Network, seven 4.7kΩ (1/7)
IC12	SN74ALS520	20	8-bit Identity Comparator
IC13	SN74ALS520	20	8-bit Identity Comparator
IC14-1	SN74ALS04B	14	Hex Inverters (1/6)
IC14-2	SN74ALS04B	14	Hex Inverters (2/6)
IC14-3	SN74ALS04B	14	Hex Inverters (3/6)
IC14-4	SN74ALS04B	14	Hex Inverters (4/6)
IC14-5	SN74ALS04B	14	Hex Inverters (5/6)
IC15-1	SN74ALS02	14	Quad 2-Input NOR Gates (1/4)
IC16-1	SN74ALS01	14	Quad 2-Input NAND Gates (1/4)
IC17	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC18-1	SN74ALS74A	14	Dual D-Type Flip-Flops (1/2)
R2		8	R Network, seven 4.7kΩ (2/7)
DL1	RWT050P	14	50ns Delay Line

[1] 300mil 24 pin DIP
[2] Single-in-line package

Fault Injection Module
Parts List (2)

Ref No.	Part Number	Size	Description
IC9-2	SN74ALS04B	14	Hex Inverters (2/6)
IC9-3	SN74ALS04B	14	Hex Inverters (3/6)
IC15-2	SN74ALS02	14	Quad 2-Input NOR Gates (2/4)
IC16-2	SN74ALS01	14	Quad 2-Input NAND Gates (2/4)
IC19	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC20	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC21	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC22	SN74ALS153	16	Dual 1 of 4 Data Selectors
R3		8	R Network, seven 4.7kΩ (3/7)
R4		8	R Network, seven 4.7kΩ (4/7)
R5		8	R Network, seven 4.7kΩ (5/7)
IC23	MC68230 P8	48	Parallel Interface/Timer (PIT-1)
IC24	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC25	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC26	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC27	MC68230 P8	48	Parallel Interface/Timer (PIT-2)
IC28	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC29	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC30	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC31	MC68230 P8	48	Parallel Interface/Timer (PIT-3)
IC32	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC33	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC34	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC35	MC68230 P8	48	Parallel Interface/Timer (PIT-4)
IC36	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC37	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC38	SN74LS449	16	Bus Transceviers w/ Bit dir.



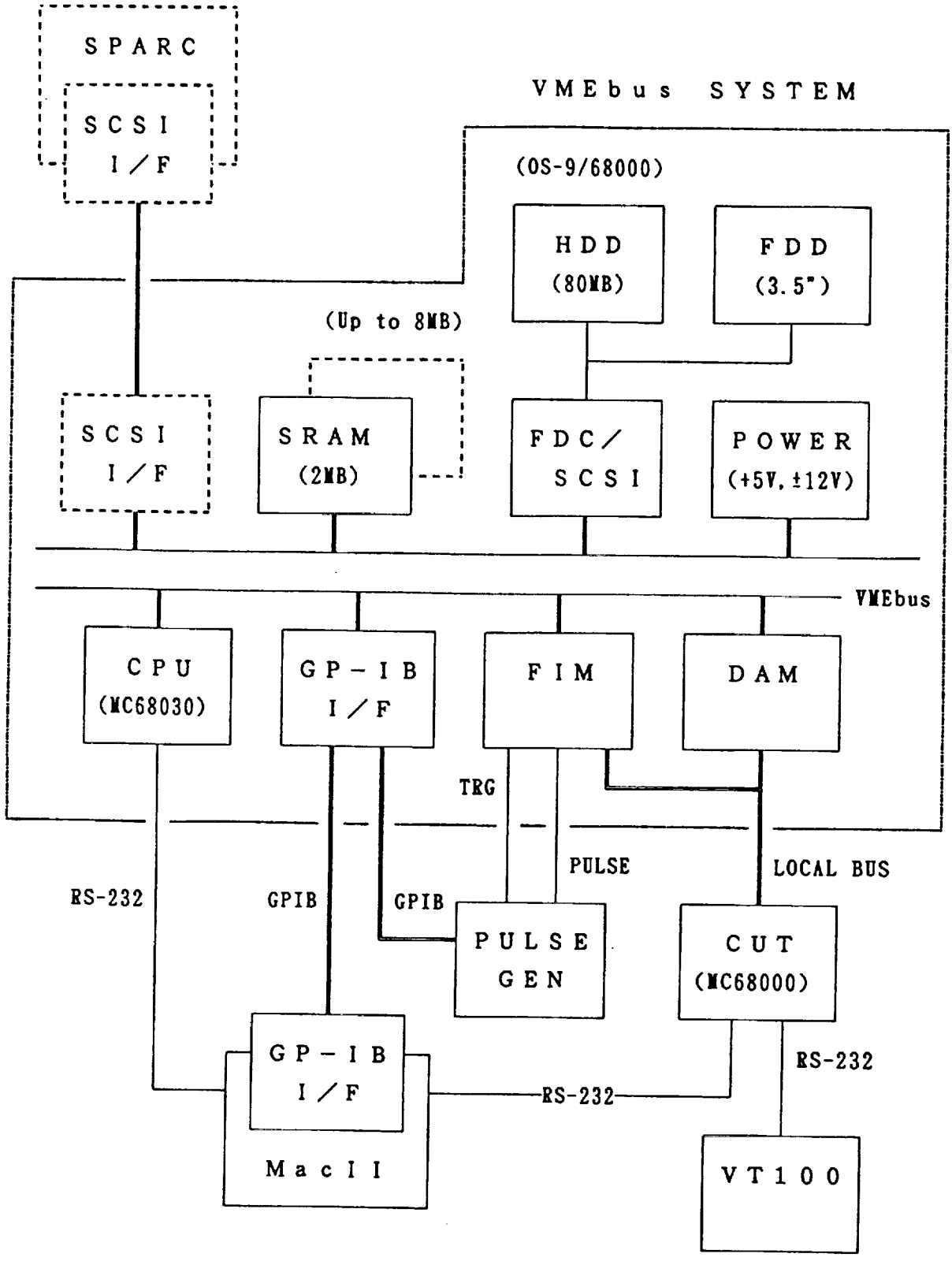
LS-446

GBA*	GAB*	DRn	OPERATION
H	H	X	ISOLATION
H	L	H	A* TO B
H	X	L	ISOLATION

FIG. FAULT INJECTION MODULE (4-BIT)

Additional Components for the New Experimental System

Part No.	Manufacturer	Description	Cost ($)
MZ 7500	MIZAR	GPIB Interface Board for VMEbus	695.00
	MIZAR	Single Cable for MZ 7500	75.00
MacII488	IOtech	GPIB Controller Board for Mac II	535.00
PFG5105	Tektronix	Pulse Generator (demo)	2,471.25
PFG5105	Tektronix	Pulse Generator (new)	2,800.75
TM5006	Tektronix	Prog. Mainframe (demo)	851.25
FIM	JHU	48ch Fault Injector	
Mac II	Apple	Macintosh II	
SPARC	Sun Micro.	SPARCstation work station	



FAULT INJECTION EXPERIMENTAL CONFIGURATION

Targeted Features of the Fault Injection Module

- **Fault Injector**
  - Provides 48 channels with bit-definable outputs using four PI/T (MC68230) and twelve bus transceiver (74LS446) chips.
  - Supports three output states (0, 1, and Z[1]) on each channel.
  - 2ch pulse generator is installed as a source of fault injections.
  - Supports single/multiple faults of stuck-at-0/1 types with duration varying from 40 ns to 99.9 ms.
- **Word Recognizer**
  - Provides a versatile trigger source for the fault injection and data acquisition.
  - Implements 16-bit word recognizer using a MC68230 PI/T and two 74LS686 magnitude comparators.

[1] Z: High-impedance

Certification Trails and Software Design for Testability

Gregory F. Sullivan[1]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Dwight S. Wilson[2]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Gerald M. Masson[3]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Abstract

This paper investigates design techniques which may be applied to make program testing easier. We present methods for modifying a program to generate additional data which we refer to as a certification trail. This additional data is designed to allow the program output to be checked more quickly and effectively. Certification trails [14, 16] have heretofore been described primarily from a theoretical perspective. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the certification trail method. The method has been applied to nine fundamental, well-known algorithms for the following problems: convex hull, sorting, huffman tree, shortest path, closest pair, line segment intersection, longest increasing subsequence, skyline, and voronoi diagram. Run-time performance data for each of these problems is given, and selected problems are described in more detail. Our results indicate that there are many cases in which certification trails allow for significantly faster overall program execution time than a 2-version programming approach, and also give further evidence of the breadth of applicability of this method.

Keywords: Software design for testability, software fault detection, certification trails, error monitoring, design diversity, data structures.

1 Introduction

We have examined a wide variety of fundamental algorithms to determine how they can be redesigned to allow for easier testability. To make the problem of testing the correctness of the output of a program more tractable we have found it is desirable to modify the program so that it generates additional data which we refer to as a *certification trail*. This additional data is designed to allow the program output to be checked more quickly and effectively. Our previous work on certification trails emphasized a theoretical perspective in which we proved that the asymptotic time complexity of the testing process could be reduced [14, 16]. In this paper, we report on implementations of the certification trail method so as to assess experimentally with run-time data the performance and overall value of the technique. We have implemented the certification trail method for nine fundamental and well-known algorithms of broad importance and applicability. For each algorithm, we have produced three implementations: a version which produces the output; a version which produces the output and generates a certification trail; and a version which checks the output while utilizing the certification trail. Specifically, algorithms for the following problems are analyzed: huffman tree, shortest path, sorting, closest pair, line segment intersection, convex hull, longest increasing subsequence, skyline, and voronoi diagram. The scope of the algorithms considered gives credibility to the overall applicability of the certification trail method. Furthermore, comparisons of run-time data for each of the three versions of each of the algorithms considered reveal many cases in which an approach using certification trails allows for significantly faster overall program execution time than a 2-version programming approach.

2 Introduction to Certification Trails

First, let us consider a basic method which is used to perform testing to detect software faults called N-version programming [1, 2]. This method utilizes N teams of programmers, each independently implementing separate programs based on a problem specification. The programs are executed on the same input and the outputs are compared. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors. Thus the technique exploits design diversity. Also, note that the method can detect hardware faults which affect the separate executions in distinct ways causing distinct outputs. It is particularly valuable for detecting errors caused by transient fault phenomena. The N-version programming method can be used to detect faults af-

[1] Research partially supported by NSF Grants CCR-8910569 and CCR-8908092 and an IBM Technology Interchange Program Grant.

[2] Research partially supported by NSF Grant CCR-8910569 and an IBM Technology Interchange Program Grant.

[3] Research partially supported by NASA Grant NSG 1442 and an IBM Technology Interchange Program Grant.

INTERNATIONAL TEST CONFERENCE 1993
0-7803-1429-8/93 $3.00 © 1993 IEEE

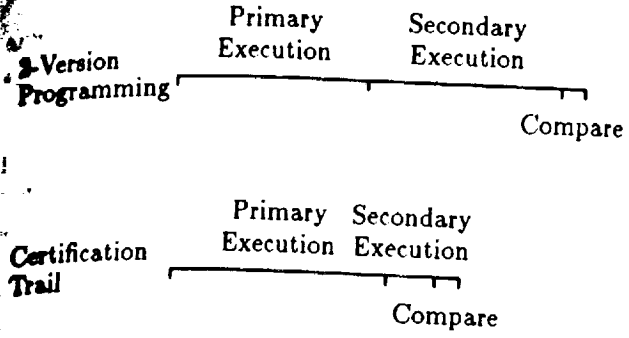


Figure 1: Timeline Comparison of the Certification Trail with 2-Version Programming

ter a system has been put into production or it can be used to detect faults in a testing phase prior to production. If two teams are used then we refer to the method as 2-version programming.

The certification-trail technique is designed to provide similar capabilities for detecting software and hardware faults as 2-version programming but expend fewer resources. As mentioned above the central idea is to modify the first algorithm so that, with modest additional overhead, it leaves behind a trail of data which we call a certification trail. This data is chosen so that it can allow the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. An illustration of typical execution times of 2-version programming versus the certification trail method is given in Figure 1. We assume that the two implementations developed for 2-version programming have approximately equal execution times. Note, however, that we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two program executions. For example, suppose the first program execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second program. It still appears possible that the execution of the second program might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first program. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected.

3 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 3.1 A problem $\mathbf{P}$ is formalized as a relation, i.e., a set of ordered pairs. Let $\mathbf{D}$ be the domain (that is, the set of inputs) of the relation $\mathbf{P}$ and let $\mathbf{S}$ be the range (that is, the set of possible solutions). We say an algorithm $\mathbf{A}$ solves a problem $\mathbf{P}$ iff for all $d \in \mathbf{D}$ when d is input to $\mathbf{A}$ then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 3.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. $\mathbf{T}$ is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ and all $t \in \mathbf{T}$ either $(F_2(d, t) = s$ and $(d, s) \in \mathbf{P})$ or $F_2(d, t) = \text{error}$.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. Intuitively, the first condition states that if both parts of our solution execute correctly, then their answers agree and are correct. The second condition states that a correct secondary execution will never produce an incorrect output, i.e., one that is not a solution to the problem.

The definitions above assure that the testing capability of the certification-trail approach is similar to that obtained with a 2-version programming approach discussed earlier. That is, if a software or hardware fault occurs during only one of the executions then either the fault will be detected or the output will be a correct solution to the problem. The examples in this paper will indicate that this new approach can save overall execution time.

4 Certification Trail Examples

In the remainder of this paper we evaluate the use of certification trails for nine classic problems in computer science. We have implemented algorithms for these problems together with other algorithms which generate and use certification trails. In addition, we

discuss a general technique for construction of certification trails for algorithms using a wide range of data structures. This technique is used to implement the certification trails for several of our examples.

We provide a full description of the algorithm for the convex hull problem which generates a certification trail and a full description of the algorithm which uses that trail. Because of space considerations the discussion of the other algorithms is abbreviated. In some cases references to previous publications or technical reports which describe the algorithms more fully are given.

The algorithms we have chosen to implement are not always the algorithms which have the smallest asymptotic time complexity. Often the asymptotically fastest algorithms have large constants of proportionality which make them slower on the data sizes we examined. We modified and used some programs from major software distributions such as quicker-sort from a Berkeley Unix distribution. Fortune's algorithm for computing the Voronoi diagram was obtained from an Internet site at AT&T Bell Labs. Other algorithms were based on textbook discussions. It should be stressed here that this research is continuing as we further increase our corpus of algorithm and data-structure implementations.

4.1 Explanation of timing data

We have collected timing data for the algorithms on a Sun SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during the timing experiments. Timing data was obtained through the getrusage() system call. The user times are reported in the data.

Much of the data presented in the timing table is essentially self-explanatory relative to the certification trail technique and algorithms considered. However, a brief discussion of the table entries is appropriate.

The column labelled *Basic* contains timing data which gives the execution time of the algorithm in producing the output without the generation of the certification trail. All timing data is listed in seconds.

The *Primary Execution (Prim. Exec.)* column gives the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Secondary Execution (Sec. Exec.)* column gives the execution time of the algorithm in producing the output while using the certification trail.

The *Percent Savings (% Sav.)* column records the percentage of the execution time savings which is gained by using the certification trail method as compared to 2-version programming approach. This assumes that both versions take approximately the same amount of time to execute.

The *Speedup* column is the ratio of the run times of the Basic Algorithm and the Secondary Execution.

For the Huffman tree data, the input size for the Huffman tree program is the number of nodes. Each node is given a frequency, chosen uniformly from the integers $\{1, 2, \ldots, n\}$. n was also selected to be the number of nodes.

For the shortest path table, there are two numbers associated with the input size, the first is the number of vertices in the graph, the second the number of edges. A graph with the required edges is selected uniformly from the set of all such graphs, then tested for connectedness in order to assure that paths exist to all vertices.

For the geometric algorithms, the input size is the number of points (or lines) in the original data set. Point set input was generated by choosing points with integer coordinates uniformly over a large square (typically 1,000,000 by 1,000,000 or larger square). For the Line Segment Intersection problem, lines were generated by picking a line segment start point uniformly from a large square and picking offsets for x and y-coordinates from a smaller range to give the end point of the line segment. This was done to bound the line length and avoid data sets resulting in a quadratic number of intersections.

Data for the longest increasing subsequence problem was produced by generating a random permutation of $[1..N]$ for input size N.

Sorting was performed on an array of pointers to structures. It was assumed that each structure contains an extra integer field for use in generating the certification trail. Sorting was performed on integer keys, though the technique can be used with a more complex key (in fact, using complex keys is very likely to increase the speedup achieved). Integers were chosen uniformly from interval $[1..1,000,000,000]$.

4.2 Convex Hull Example

The convex hull problem is fundamental in the field of computational geometry. Our certification trail solution is based on a convex hull algorithm due to Graham [6] called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos[11]. For simplicity in the discussion which follows we will assume the points are in general position, e.g., no three points are collinear. It is not hard to remove this restriction.

Definition 4.1 The *convex hull* of a set of points, T, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique

and its vertices are a subset of the points in T. It is specified by a counterclockwise sequence of its vertices.

The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. The first step of the algorithm selects an "extreme" point and calls it p_1. The next two steps sort the remaining points. The order of the points is determined by the slopes of the line segments formed by joining each point to p_1. It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although this polygon may not be convex. The Graham Scan algorithm traverses this polygon, removing points until the resulting polygon is convex. The main FOR loop iteration adds vertices to the polygon under construction and the inner WHILE loop removes vertices from the construction. A point is removed when the angle test performed at line 6 reveals that the angle at that vertex is obtuse. It is easy to demonstrate that when a point is removed, it must fall within the triangle defined by three other points, p_1 and the two points that were adjacent to the point removed. When the main FOR loop is complete the convex hull has been constructed. The execution of this algorithm is demonstrated in Figure 2. For each removed point, the associated triangle is indicated in bold lines, and in the text below the diagram. Our certification trail relies on the fact that that these triangles can be determined quickly.

Algorithm CONVEXHULL(T)
Input: Set of points, T, in R^2
Output: Counterclockwise sequence of points in R^2 which define the convex hull of T
1 Let p_1 be the point with the largest x coordinate (and smallest y to break ties)
2 For each point p (except p_1) calculate the slope of the line through p_1 and p
3 Sort the points (except p_1) from smallest slope to largest. Call them $p_2, \ldots, p_n$
4 $q_1 := p_1$; $q_2 := p_2$; $q_3 := p_3$; $m = 3$
5 FOR $k = 4$ to n DO
6 WHILE the angle formed by q_{m-1}, q_m, p_k is ≥ 180 degrees DO $m := m - 1$ END
7 $m := m + 1$
8 $q_m := p_k$
9 END FOR
10 FOR $i = 1$ to m DO, OUTPUT(q_i)
END FOR
END CONVEXHULL

First execution: In this execution the code CONVEXHULL is used. The certification trial is generated

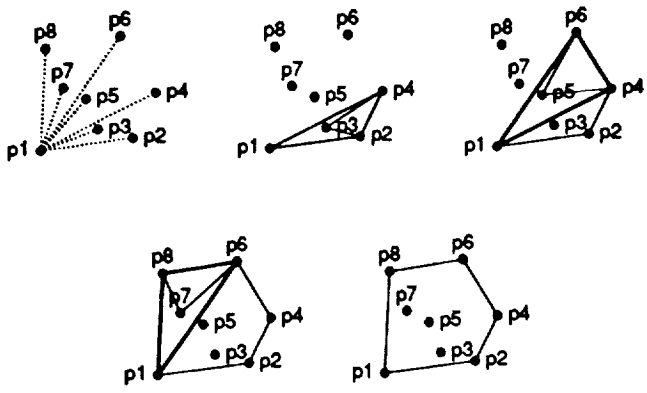


Figure 2: Convex hull example.

Point not on convex hull	Three surrounding points
p_3	p_1, p_2, p_4
p_5	p_1, p_4, p_6
p_7	p_1, p_6, p_8

by adding an output statement within the WHILE loop. Specifically, if an angle of less than 180 degrees is found in the WHILE loop test then the four tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Strictly speaking the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $p_1, p_2, \ldots, p_n$ then rather than output the element in R^2 corresponding to p_i the number i is output.

Second execution: Let the certification trail consist of a set of four tuples, $(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r)$ followed by the supposed convex hull, $q_1, q_2, \ldots, q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

- First, it checks that there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.
- Second, it checks that for each $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.
- Third, the algorithm checks that for each $i \in \{1, \ldots, r\}$, x_i lies within the triangle defined by a_i,b_i, and c_i.

- Fourth, the algorithm checks that for each triple of counterclockwise consecutive points on the supposed convex hull, the angle formed by the points is less than or equal to 180 degrees.

- Fifth, it checks that there is a unique point among the points on the supposed convex hull which is a local maxima. We say a point q on the hull is a *local maxima* if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. Otherwise the convex hull read from the trail is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; instead it makes them easier. The correctness and adequacy of these checks must be proven. A complete formal proof is beyond the scope of this paper, instead a brief outline of the proof will be given.

Using our formal definition of certification trails, let $\mathbf{D}$ be the set of all finite planar point sets T. Let $\mathbf{S}$ be the set of convex polygons, with vertices in counterclockwise order (the restriction to counterclockwise ordering makes the convex hull unique). Then the problem we are considering is $HULL : \mathbf{D} \rightarrow \mathbf{S}$ where $HULL(T)$ is the polygon in $\mathbf{S}$ that forms the convex hull of T.

The description of the algorithms above defines functions F_1 and F_2. We must show that both conditions of Definition 3.2 hold. The following two lemmas, which we state without proof, are required.

Lemma 4.2 *Let P be a polygon on n points $p_1, p_2, \ldots, p_n$. P is a convex polygon iff P is simple and each angle $p_i p_j p_k$ is less than or equal to 180 degrees, where i is in $1, 2, \ldots n$, $j = (i+1) \bmod n$, and $k = (i+2) \bmod n$.*

Lemma 4.3 *If P is a non-simple polygon, then either P has more than one local maxima, or the interior angle at some vertex is greater than 180 degrees.*

These are deceptively simple statements. Though they are intuitively obvious, a formal proof is difficult. It is interesting to note that some computer graphics texts give an incorrect test for determing convexity of a polygon by omitting the check for simplicity required by Lemma 4.2.

Recall that the first condition is:

For all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$.

Intuitively, this means that if both executions perform correctly then they will both output the convex hull of the input, which is unique. Note that generation of the certification trail does not affect the output of the Graham Scan algorithm. Thus the condition on $F_1(d)$ is satisfied by the correctness of the Graham Scan algorithm, the proof of which is well known [11]. To show that $F_2(d,t) = s$, note that a copy of s is contained on the trail t. Our description of $F_2(d,t)$ states that s is output unless one of the five checks above fails. It is trivial to verify that the first three of these checks must be satisfied. The fourth check cannot fail, since the polygon described by s is convex (because $(d,s) \in \mathbf{P}$). Similarly, if the fifth check fails, then the polygon described by s has two local maxima, and this is not possible for a convex polygon.

The second condition is:

For all $d \in \mathbf{D}$ all $t \in \mathbf{T}$ either ($F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$) or $F_2(d,t) =$ error.

Intuitively, this means that given an input and arbitrary trail, $F_2(d,t)$ produces a solution to the problem or flags an error.

Our definition of $F_2(d,t)$ states that the polygon Q stored on the trail is output unless one of the five checks fails. We must therefore demonstrate that if all five checks succeed, then Q is the convex hull of the input points d. Let H be the convex hull of the points d, The first condition guarantees that every point in d is classified as a hull point or an interior point. The second condition guarantees that the triangles used to identify interior points are formed from input points, and the third check verifies that the interior points are indeed inside their respective triangles. Note that we do not attempt to verify that the triangles used are the ones that would be produced by $F_1(d)$. In general, for a given interior point, there may be several triangles of input points in which it is contained. Together, the first three conditions imply that all points in H are also in Q, since it is impossible for a hull point to be contained in a triangle. Note that these three checks do not exclude the possibility that interior points are present in Q, nor do they guarantee that the ordering of the hull points in Q is correct. The final two checks will accomplish this. If the last two checks are satisfied, Lemma 4.3 states that Q is simple, and therefore it must be convex by Lemma 4.2.

Thus, Q is a convex polygon whose vertex set is a superset of the vertices of H, i.e., H is contained in T. This implies that no other point from the input set may be a vertex of Q, since any input point that is not a hull point is interior to H and therefore interior to Q. Finally, it is clear that the ordering of the vertices of Q and H must be the same (although there

might appear to be two possible orderings, clockwise and counterclockwise, a clockwise ordering will fail the fourth check). Therefore if all five checks succeed, then the output of $F_2(d,t)$ will be the convex hull of d.

This demonstrates that the algorithms described meet the conditions of Definition 3.2, and are therefore a certification trail solution to the convex hull problem.

Time complexity: In the first execution the sorting of the input points takes $O(n\log(n))$ time where n is the number of input points. One can show that this cost dominates and the overall complexity is $O(n\log(n))$.

It is possible to implement the second execution so that all five checks are done in $O(n)$ time. The first two checks may be done in linear time since the certification trail contains indices into the input data. The third and fourth checks require a constant time calculation at each point. Finally, the uniqueness of the local maxima is clearly checkable in linear time.

Order-of-Magnitude Testing Speedup: It should be noted that for the convex hull problem, we are seeing an order of magnitude speedup for reasonable sized problems. We believe this offers a dramatic demonstration of the efficiency of our proposed software testing technique using certification trails in comparison with the 2-version programming technique.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
5000	0.64	0.67	0.08	41.41	8.00
10000	1.38	1.40	0.17	43.12	8.12
25000	3.89	3.84	0.46	44.73	8.46
50000	8.44	8.50	0.85	44.61	9.93
00000	17.36	17.68	1.65	44.33	10.52

Table 1: Convex Hull

4.3 Sorting Example

This important problem has a massive literature. In this section we will discuss how to apply the certification trail approach to the sorting problem. Let us assume that the sorting algorithm takes as input an array of n elements and outputs an array of n elements. The algorithm is supposed to place the data in non-decreasing order.

To design a certification trail algorithm we must discover the nature of the data that should be included in the certification trail to allow quick computation of the final output sorted array. Suppose that we decide to use the output array itself as the certification trail. We note that it is easy to check that this array is in non-decreasing order by simply performing a single pass over the array. Unfortunately, it is considerably more difficult to make sure that this array contains exactly the same elements as the original input array. Indeed, this problem has a lower bound time complexity of $\Omega(n\log(n))$ in a comparison based model.

Because of this difficulty we use the permutation of the elements defined by the input and output data arrays as the certification trail. This permutation is computed by attaching an Item Number field to the data elements before sorting. The i-th item receives item number i. After the elements are sorted, the permutation from input to output is obtained by reading the Item Numbers from the elements in their new order.

The second execution reads the permutation from the trail and verifies that it is a permutation on n elements, i.e., that no numbers are repeated or omitted. This permutation is used to rearrange the input elements in linear time. Finally the algorithm checks that these elements are now in non-decreasing order.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
10000	0.28	0.30	0.04	39.29	7.00
50000	1.80	1.90	0.19	41.94	9.47
100000	3.96	4.08	0.41	43.31	9.66
500000	23.95	24.69	2.14	43.99	11.19
1000000	50.23	51.57	4.38	44.31	11.47

Table 2: Sort

4.4 Certification Trails For Abstract Data Types

Before we present the rest of our example algorithms we discuss a general technique applicable to many algorithms and data structures.

An *abstract data type* is a data object or set of data objects together with a group of operations for manipulating the object(s). Each operation takes a (possibly empty) set of arguments, and some, but not necessarily all, operations return answers. Many algorithms make extensive use of abstract data types.

We describe a method for automatically generating a certification trail for an algorithm which uses an abstract data type. This is done by modifying the abstract data type operations, so that during the first execution they generate a certification trail, and during the second execution they use the certification trail. Otherwise, these operations are identical to the original abstract data type operations, i.e., they take the same type of arguments and have the same return types. The object of creating and using the certification trail is to

allow a more efficient implementation of the abstract data type during the second execution.

We illustrate this technique for the following abstract data type which we call *Ordered Collection*. An Ordered Collection will contain a set of pairs (i, x) where i is an item number, and x is a real number value. (This selection is made for simplicity of description, the elements being stored could be more complex). No two elements of the set may have the same item number, though several items may have a common value. We define a total ordering on pairs by $(i, x) < (i', x')$ iff $x < x'$ or $x = x'$ and $i < i'$

The following operations are defined on an Ordered Collection:

INSERT(i,x) Add the element (i, x) to the set.

DELETE(i) Delete the element with item number i from the set.

PREDECESSOR(i) Let (i, x) be the element in the set with item number i. This operation returns its predecessor, that is, the largest pair less than (i, x). A special value SMALLEST is returned if (i, x) is the smallest element in the set.

MIN Return the smallest element in set.

NEAREST(x) Return the element from the set with value closest to x. If there is a tie, return the element with the smallest item number.

This small set of operations is being chosen for concreteness, several additional operations could be easily defined. If an error occurs during any of these operations, for example, inserting pairs with duplicate item numbers or attempting to delete a non-existent item, then the program terminates indicating an error.

These operations may be modified to produce a certification trail during the first execution by modifying the INSERT(i,x) and NEAREST(x) operations to do the following (in addition to their normal function):

INSERT(i,x) After adding this element to the set, perform a PREDECESSOR(i) operation and write the item number of the answer to the certification trail.

NEAREST(x) Write the item number of the answer to the certification trail.

A typical implementation of an abstract data type supporting the above operations would require $\Omega(n \log(n))$ time to process a sequence of n operations. By using the certification trail, we can achieve linear time for n operations during the second execution. This includes the time necessary to check the trail for correctness as well as use it.

The implementation of the Ordered Collection for the second execution will be a structure called an indexed linked list. This is a doubly linked list, along with an array *Items* of pointers, indexed by item number. The i-th element in this array points to the list node for the element with item number i (or is NULL if no element in the list has item number i). This allows us to find an element in constant time given its item number. The elements themselves are maintained in ascending order (according to the pair ordering given above) on a doubly linked list, i.e., each element has pointers to its successor and predecessor. In addition to the array, we maintain a variable *Start*, which stores the item number of the first element in the list.

The abstract data type operations for the second execution are defined as follows:

INSERT(i,x) Read the item number p from the trail. p is the item number that would be the predecessor of (i, x) if it were in the set. *Items*[p] points to the list node for the element with index p, call this element (p, x_p). We can insert (i, x) after this node using ordinary list operations. Before doing so, however, we make three checks:

i. Check that *Items*[i] is currently NULL, i.e., there is not currently an element with item number i in the set.

ii. Check that (i, x) is greater than (p, x_p).

iii. Check that (i, x) is less than the successor of (p, x_p)

If these checks are satisfied, then (i, x) may be inserted after (p, x_p). Set *Items*[i] pointing to the list node for (i, x).

Note that special cases occur at the beginning and end of the list. We omit the specifics of these cases, mentioning only that *Start* must be updated for insertions at the front of the list.

DELETE(i) Check that *Items*[i] is not NULL, i.e., there is an element with item number i currently in the set. If so, remove it from the linked list, and set *Items*[i] to NULL. If we remove the first element of the list we must also update *Start*.

PREDECESSOR(i) *Items*[i] points to the element with item number i, and its predecessor may be found by following the appropriate pointer.

MIN The variable *Start* indicates the item number of the first element on the list, i.e., the minimum element. *Items*[*Start*] therefore points to this element.

NEAREST(x) Read the index i from the trail. *Items*[i] points to the element having this item number, call it (i, v). To verify that this is the correct answer we will have to check one of its neighbors. If $v < x$, then only the successor of (i, x) could have a value closer to v. Otherwise, only the predecessor is a candidate. Check the appropriate neighbor.

Although our example uses elements that contain item numbers, it is not necessary that the abstract data type be defined in this way. The insert operation of an abstract data type may be modified to tag elements with item numbers as they are inserted.

Variations on this scheme are possible. For example, by modifying DELETE(i) and NEAREST(x) operations so that they also write the item numbers of predecessors to the trail, it is possible to use a singly linked list during the second execution. More sophisticated schemes, involving marking list nodes for deletion and delayed checks, allow the use of singly linked lists without requiring DELETE(i) and NEAREST(x) to produce predecessor information.

The technique in this example generalizes to other abstract data types supporting a predecessor operation. In fact, a somewhat weaker condition often suffices; it is sufficient that the specific implementation of the abstract data type allow the predecessor of an element to be found at the time the element is inserted. The abstract data type itself need not support a predecessor operation. This technique is used in four of our example algorithms.

Using this technique, it is possible to reuse the first execution code, except for the code implementing the abstract data type operations. One advantage of this is that it may be possible to add extra checking to such code, such as bounds checking and checks on pointer references, that may be too expensive to include in the first execution. Of course, the two programs may be developed separately as long as the specifications agree on the use of the abstract data type.

Space does not permit a full proof of correctness of this scheme. A proof proceeds by establishing the following invariants on the indexed linked list used in the second execution.

i. The pairs in the linked list are in order from smallest to largest.

ii. Each element of the *Items* array is either NULL or points to one of the nodes in the linked list.

iii. If *Items*[i] is not NULL, then the list node pointed to by it stores an element with item number i. (Note that this implies that each list node is pointed to at most once).

iv. Every node in the list is pointed to by some item in *Items*[i].

v. *Start* is the item of the first element in the list.

These conditions are clearly satisfied by an indexed linked list containing no elements (i.e., before any operations have been performed). Inspection of operations that query the list (MIN and NEAREST for example) shows that they function correctly if the above conditions are met. It is easy to prove correctness of the certification trail by demonstrating that the operations maintain a one to one corresponce between the pairs in the linked list and the elements in the abstract data type and that the above invariants are preserved.

4.5 Shortest Path Example

This is another classic problem which has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [3] as explicated in [17]. We are concerned with the single source problem, i.e., given a graph and a vertex s, find the shortest path from s to v for every vertex v.

The algorithm for this problem which has the fastest asymptotic time complexity uses fusion trees and is given in [5]. This algorithm however appears to have a large constant of proportionality and therefore we do not use it.

We use the techniques just discussed to implement the certification trail for this problem. A full description may be found in a technical report [15].

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
100,1000	0.04	0.05	0.02	12.50	2.00
250,2500	0.15	0.16	0.06	26.67	2.50
500,5000	0.31	0.33	0.11	29.03	2.82
1000,10000	0.70	0.76	0.23	29.29	3.04
2000,20000	1.58	1.67	0.45	32.91	3.51
2500,25000	2.06	2.15	0.55	34.47	3.75

Table 3: Shortest Path

4.6 Huffman Tree Example

This is another classic algorithmic problem and one of the original solutions was found by Huffman[7]. It has been used extensively to perform data compression through the design and use of so called Huffman codes. These codes are prefix codes which are based on the

Huffman tree and which yield excellent data compression ratios. The tree structure and the code design are based on the frequencies of individual characters in the data to be compressed. Here we are concerned exclusively with the Huffman tree. See [7] for information about the coding application.

Definition 4.4 The Huffman tree problem is the following: Given a sequence of frequencies (positive integers) $f[1], f[2], \ldots, f[n]$, construct a tree with n leaves and with one frequency value assigned to each leaf so that the weighted path length is minimized. Specifically, the tree should minimize the following sum: $\sum_{l_i \in \text{LEAF}} \text{len}(i) f[i]$ where LEAF is the set of leaves, $\text{len}(i)$ is the length of the path from the root of the tree to the leaf l_i, $f[i]$ is the frequency assigned to the leaf l_i.

A full description of the method we employ to generate and use a certification trail is detailed in a technical report [15].

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
5000	0.81	0.87	0.16	36.42	5.06
10000	1.76	1.86	0.33	37.78	5.33
25000	6.01	6.30	1.02	39.10	5.89
50000	10.62	11.14	1.70	39.55	6.25

Table 4: Huffman tree

4.7 Other problems

We report timing data for five other problems, the "Manhattan skyline" problem, computation of Voronoi diagrams, longest increasing subsequence, the closest pair problem, and line segment intersection. Space permits only a brief description of these problems, rather than a full exposition of the certification trail techniques used.

The "Manhattan skyline" problem is: Given a set of rectangles with collinear bottom edges, compute the polygonal outline of the union of the rectangles [9].

The Voronoi diagram is a fundamental concept in computational geometry [11]. Given a set of points P in the plane, the Voronoi diagram is a partition of the plane into regions such that each region consists of all points closer to a given $p \in P$ than to any other other point in P. Computation of the Voronoi diagram is an important step in many problems involving point location.

The next problem we consider is, given a sequence of integers, find the longest (not necessarily unique) strictly increasing subsequence.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
1000	0.27	0.26	0.12	29.63	2.25
5000	1.69	1.65	0.57	34.32	2.96
10000	3.91	3.72	1.14	37.85	3.43
15000	6.08	5.78	1.77	37.91	3.44
20000	8.53	8.27	2.33	37.87	3.66

Table 5: Skyline

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
100	0.04	0.04	0.03	12.50	1.33
500	0.24	0.26	0.19	6.25	1.26
1000	0.51	0.51	0.39	11.76	1.31
5000	2.75	2.82	2.03	11.82	1.35
10000	5.79	5.89	4.06	14.08	1.43
50000	40.15	40.63	22.00	22.00	1.83

Table 6: Voronoi Diagram

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
10000	0.13	0.14	0.04	30.77	3.25
50000	0.78	0.81	0.22	33.97	3.55
100000	1.61	1.70	0.44	33.54	3.66
500000	9.17	9.32	2.22	37.08	4.13
1000000	18.66	19.58	4.46	35.58	4.18

Table 7: Longest Increasing Subsequence

Given a set of points P in the plane, the Closest Pair problem is that of finding the pair of points with minimum distance over all pairs in the set.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
10000	0.26	0.27	0.07	34.62	3.71
50000	1.45	1.55	0.36	34.14	4.03
100000	3.06	3.26	0.72	34.97	4.25
500000	16.84	18.02	3.62	35.75	4.65

Table 8: Closest Pair

Given a set of line segments in the plane, the line intersection problem is the problem of determining all intersections of line segments in this set.

For the first four problems, algorithms running in $O(n \log(n))$ time were implemented for the first execution. The second execution, using certification trails, runs in linear time. The first execution algorithm used for line intersection runs in $(O((k+n)\log(n))$ time where k is the number of intersections and n the number of points. The second execution runs in $O(k+n)$ time. Note that k may be quadratic in n.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
1000	0.47	0.49	0.04	43.62	11.75
2500	1.45	1.53	0.12	43.10	12.08
5000	3.33	3.47	0.26	43.99	12.81
10000	7.72	7.88	0.60	45.08	12.87
25000	24.00	24.12	1.75	46.10	13.71

Table 9: Line Segment Intersection

5 Concluding Discussion

Certification trails have heretofore been discussed principally from a theoretical perspective. In this paper we have presented experimental timing data which illustrates the advantages of the certification trail technique for software testing over the 2-version programming technique. We have further presented techniques and analytical results for several new algorithms which further support the significance of the certification trail technique by demonstrating its broadening applicability. It should be appreciated that the scope of our experimental investigation is not limited to the algorithms considered here; numerous other algorithms we have considered could have been discussed, and we continue to work on new applications. It should also be pointed out that in addition to the timing experiments reported here, software fault injection experiments have also been conducted which verify the detection capabilities of the certification trail method. The breadth of applicability of the certification trail technique continues to expand along with the credibility of its advantages. Increasingly, the certification trail method can be viewed as a competitive software testing alternative.

References

[1] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[2] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *1978 Fault Tol. Comp. Symp.*, pp. 3-9, IEEE Computer Society Press, 1978.

[3] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, 1959.

[4] Fortune, S. "A Sweepline Algorithm for Voronoi Diagrams," *Algorithmica*, pp. 153-174, 2, 1987.

[5] Fredman, M. L., and Willard, D. E., "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st IEEE Foundations of Computer Science*, pp. 719-725,1990.

[6] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set", *Information Processing Letters*, pp. 132-133, 1, 1972.

[7] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[8] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[9] Manber U., *Introduction to Algorithms* Addison-Wesley, Reading, MA, 1989.

[10] Nievergelt, J., and Hinrichs, K. H., *Algorithms and Data Structures With Applications to Graphics and Geometry*, Prentice Hall, NJ 1993

[11] Preparata F. P., and Shamos M. I., *Computational geometry*, Springer-Verlag, New York, NY, 1985.

[12] Sedgewick, R., "Implementing quicksort programs," *Comm. of the ACM*, pp. 847-857, 21(10), 1978.

[13] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[14] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[15] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Department of Computer Science Technical Report JHU 89/26*, Johns Hopkins University, Baltimore, Maryland, 1989.

[16] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Digest of the 1991 Fault Tolerant Computing Symposium*, pp. 240-247, IEEE Computer Society Press, 1991.

[17] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

Experimental Evaluation of Certification Trails using Abstract Data Type Validation

Dwight S. Wilson[1]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Gregory F. Sullivan[2]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Gerald M. Masson[3]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [11, 12]. Recent experimental work [13] reveals many cases in which a certification-trail approach allows for significantly faster program execution time than a basic time-redundancy approach. Algorithms for answer-validation of abstract data types are presented in [12] and allow a certification trail approach to be used for a wide variety of problems. In this paper, we report on an attempt to assess the performance of algorithms utilising certification trails on abstract data types. Specifically we have applied this method to the following problems: heapsort, Huffman tree, shortest path, and skyline. Previous results used certification trails specific to a particular problem and implementation. The approach in this paper allows certification trails to be localized to "data structure modules," making the use of this technique transparent to the user of such modules.

Keywords: Software fault tolerance, certification trails, error monitoring, design diversity, data structures.

1 Introduction

To explain the essence of the certification trail technique for software fault tolerance, we first discuss 2-version programming [4, 2]. Using 2-version (or more generally, N-version) programming, two (or N) implementations of an algorithm are executed on a given input, and the results compared. If the outputs agree, they are accepted, otherwise an error is flagged. This technique will detect a variety of software faults as well as transient hardware faults. A variation of this technique is to execute a single program twice and compare results, this is called time redundancy. Although there are a few software faults that may be detected using time redundancy (e.g., uninitialised pointer errors), it is more effective in catching transient faults.

The certification trail technique is designed to achieve similar types of error detection capabilities but expend fewer resources. The central idea, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. The second algorithm may then make use of this data, which is chosen so that the algorithm executes more quickly and/or has a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains a error which causes an incorrect output and an incorrect certification trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the data trail.

Early work on the certification trail focused on creating trails for specific implementation of problems. For example the trail given in [11] for the convex hull problem is specific to the Graham scan algorithm. In general, the two algorithms used in this approach can be quite different. A more recent approach is to construct a certification trail for an abstract data type. That is, given the answers to operations allowed on that type, our algorithm checks the correctness of these answers. This method has the advantage that the certification trail techniques are localised to the

[1]Research partially supported by NSF Grants CCR-8910569 and IBM Technology Interchange Program Grant.
[2]Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.
[3]Research partially supported by NASA Grant NSG 1442.

outines implementing data structure operations, and nay then be applied to a wide variety of problems without special coding. In many cases it may be possible to use existing code with only minor modifications. Code using these routines is run twice, the first ime generating the trail, the second time using it. Alternately, the trail checking may be done, in parallel, .e., we perform the checking as the trail is being gen-rated. A programmer using a library of these routines need not be familiar with certification trail techniques. Object oriented programming techniques may be par-icularly useful for implementation of such "certified" data types.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realsations and uses.

Definition 2.1 A problem P is formalised as a relation, i.e., a set of ordered pairs. Let D be the domain (that is, the set of inputs) of the relation P and let S be the range (that is, the set of solutions) for the problem. We say an algorithm A solves a problem P iff for all $d \in \mathbf{D}$ when d is input to A then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 2.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. T is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that
$F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$

(2) for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$
either ($F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$)
2 or $F_2(d, t) =$ error.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

3 Answer Validation Problem for Abstract Data Types

Our general approach to applying certification trails uses the concept of an abstract data type. Some examples of abstract data types are given later in this paper. Here we mention some important common properties and give a short illustration. Each abstract data type has a well defined data object or set of data objects. Each abstract data type has a carefully defined finite collection of operations that can be performed on its data object(s). Each operation takes a finite number of arguments (possibly zero). In addition, some but not all operations return answers. An example of an abstract data type is a priority queue. The data object for a priority queue is an ordered pair of the form (i,k) where i is an item number and k is a key value. A priority queue has two operations: insert(i,k) and delmin. The insert operation has two arguments: item number i and key value k. The insert operation does not return an answer. The delmin operation has no arguments, but it does return an answer. The precise semantics of these operations are given later in this paper.

For each abstract data type we may define an *answer validation* problem. Intuitively, the answer validation problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on the abstract data type. More formally, the input to the answer validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. In addition, the sequence contains the supposed answers for each of the operations which return answers. In particular, each supposed answer is paired with the operation that is supposed to return it.

The output for the answer validation problem is the word "correct" if the answers given in the input match the answers that would be generated by actually performing the operations. The output is the word "incorrect" if the answers do not match. It is also useful to allow the output word to say "ill-formed". This output is used if the sequence of operations is ill-formed, e.g., an operation has too many arguments or an argument refers to an inappropriate object.

The answer validation problem is similar to the idea

of an acceptance test which is used in the recovery block approach [10] to software fault tolerance. The main difference is that an answer validation problem is dependent upon a sequence of answers, not just an individual answer. Hence, if an incorrect answer appears in the sequence, it may not be detected immediately. It is guaranteed, however, that an incorrect will be detected at some point during the processing of the entire sequence. By allowing for this latency in detection, it is possible to create a much more efficient procedure for solving the answer validation problem.

The most important aspect of the answer validation problem is the fact that is is often possible to check the correctness of the answers to a sequence of operations much more quickly than actually calculating what the answers should be from scratch. In other words, the answer validation problem has a smaller time complexity than the original abstract data type problem. For example, to calculate the answers to a sequence of n priority queue operations takes $\Omega(n \log(n))$ time in the decision tree model; however, it is possible to check the correctness of the answers in only $O(n)$ time [12]. This speed is very useful in fault-detection applications.

It is possible to run an answer validation algorithm for some abstract data type concurrently with some algorithm which uses the abstract data type. The answer validation algorithm could act as a monitor making sure that all interactions with the abstract data type are handled correctly. This is valuable because many algorithms spend a large fraction of their time operating on abstract data types. Note, the overhead of this monitor is less than the overhead of actually performing the data type operations twice.

4 Schema for using Certification Trails

Suppose that we have developed an efficient solution to the answer validation problem for some abstract data type. By efficient we mean the time complexity of the answer validation problem is smaller than the time complexity of the original abstract data type problem. Further, suppose that we wish to run an algorithm, say A, which uses that abstract data type. To apply the certification trail method we can use the following schema to yield the two executions:

First Execution:

Execute algorithm A.
Each time an abstract data type operation is performed. Append to the certification trail the identity of the operation, the arguments and the answer.

Second execution:

Phase One:
Validate the correctness of the operations and supposed answers given in the certification trail. If the validation returns "incorrect" or "ill-formed" then output "error" and stop. Otherwise, continue.

Phase Two:
Execute algorithm A.
Each time an abstract data type operation is performed. Read the next entry in the certification trail. Make sure that the operation and the arguments in the certification trail agree with those requested in the algorithm. If not output "error" and stop. Otherwise, use the answer given in the certification trail and continue.

This schema can yield execution times which are significantly faster than the execution time obtained by running algorithm A twice. Yet the schemes yield comparable fault detection capabilities. Note, the first execution can be slower than a simple execution of algorithm A since it must output a certification trail. However, the second execution can be significantly faster than a simple execution of the algorithm since the interactions with the abstract data type take less time overall. The net effect can yield a major speed-up.

Suppose an algorithm uses multiple abstract data types and suppose there are efficient answer validation algorithms for each of these abstract data types. It is easy to see how our method generalizes. We can leave behind a generalised certification trail which consists of a seperate certification trail for each of the abstract data types. The effect on the speed up of the second execution will be cumulative.

5 Generalized Priority Queue

We now describe a somewhat general abstract data type. We are able to solve the answer validation problem for restricted versions of this data type. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is

an item number, k is a key value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i,k) < (i',k')$ iff $k < k'$ or $(k = k'$ and $i < i')$. The abstract data types we will consider support a subset of the following operations.

member(i) returns a boolean value of true if the set contains an ordered pair with item number i, otherwise returns false.

insert(i,k) adds the ordered pair (i,k) to the set. We require that no other pair with item number i be in the set.

delete(i) deletes the unique ordered pair with item number i from the set. We require that a pair with item number i be in the set initially.

changekey(i,k) is executed only when there is an ordered pair with item number i in the set. This pair is replaced by (i,k).

deletemin returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If the set is empty then the token "empty" is returned.

min returns the ordered pair which is smallest according to the total order defined above. If the set is empty then the token "empty" is returned.

max and deletemax these operations are similar to min and deletemin, using the largest element instead of the smallest one.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

Many different types and combinations of data structures can be used to support different subsets of these operations efficiently. Specifically we are interested in allowing the insert, delete, min, and deletemin operations. It is possible to process a sequence of $O(n)$ operations in $O(n\log(n))$ with implementations using heaps or balanced search trees such as AVL trees [1], red-black trees [6] or b-trees [3]. Answer validation of these operations can be performed in $O(n)$ time [12, 13].

6 Examples of the use of Data Structure Certification

In this section we evaluate the use of certification trails for data structures as applied to four well-known and significant problems in computer science: sorting, the shortest path tree problem, the Huffman tree problem, and the skyline problem. We have implemented basic algorithms for these problems and algorithms which generate and use certification trails. Timing data was collected using a SPARCstation ELC.

The timing information reported in the tables consists of the run time of the basic algorithm (i.e., no certification trail), the run time of the trail-generating algorithm, the run time of the trail-using algorithm, the percentage savings of using certification trails, and the speedup achieved by the second phase of the certification trail method. The percentage savings is computed by comparing the total run time of algorithms for generating and using trails against twice the run time of the basic algorithm. The speedup is computed by dividing the run time of the basic algorithm by the run time of the algorithm that uses the certification trail.

Apart from the data structures, the implementation of both phases of the certification trail version of each algorithm is nearly identical to the implementation of the basic version. The only difference in the code for the two phases is a parameter passed to the data structure code indicating whether a certification trail should be generated or used. All code implementing the certification trails is localised to the modules implementing the data structures, allowing the generation and use of the trail to be transparent to the user of these modules. Due to space constraints only an abbreviated discussion of the algorithms is given.

6.1 Heapsort

Sorting is a fundamental operation in computer systems, and there exist several sorting algorithms. Sorting may be implemented with a priority queue (or more specifically, a heap) by inserting all elements and performing deletemin operations until the queue is empty.

Input data was generated by creating sets of integers chosen uniformly from the interval $[0, 10000000]$. Timing results are based on fifty executions at each input size.

6.2 Huffman Tree

Given a sequence of frequencies (positive integers), we wish to construct a Huffman tree, i.e., a binary tree with frequencies assigned to the leaves, such that the sum of the weighted path lengths is minimised. This is a classic algorithmic problem and one of the original solutions was found by Huffman [7]. It has been used

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
10000	0.44	0.45	0.11	86.36	4.00
20000	0.98	1.00	0.23	87.24	4.26
50000	2.71	2.80	0.60	87.27	4.52
100000	5.87	6.05	1.23	87.99	4.77
200000	12.71	13.91	2.47	89.50	5.15
500000	19.67	20.25	3.73	89.04	5.27

Table 1: Heapsort

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
5000	0.38	0.41	0.14	27.63	2.71
10000	0.83	0.87	0.29	30.12	2.86
20000	1.79	1.90	0.61	29.89	2.93
50000	4.93	5.30	1.53	30.73	3.22
100000	10.75	11.47	3.13	32.14	3.45
150000	16.70	17.87	4.66	32.54	3.58

Table 2: Huffman Tree

extensively in data compression algorithms through the design and use of so called Huffman codes. The tree structure and code design are based on frequencies of individual characters in the data to be compressed. In this paper we are concerned only with the Huffman tree, the interested reader should consult [7] for information about the coding application.

The Huffman tree is built from the bottom up and the overall structure of the algorithm is based on the greedy "merging" of subtrees. An array of pointers, ptr, is used to point to the subtrees as they are constructed. Initially, n single vertex subtrees are constructed, each one associated with a frequency number in the input. The algorithm repeatedly merges the two subtrees with the smallest associated frequency values, assigning the sum of these frequencies to the resulting tree. A priority queue data structure allows the algorithm to quickly find the subtrees to merge at each step.

Data for the timing experiments was generated by choosing integer frequencies uniformly from the range $[0, 100000]$. Timing results are based on fifty executions for each input size.

6.3 Shortest Path

Given a graph with non-negative edge weights and a source vertex, we wish to find the shortest paths from the source vertex to each of the other vertices. This is another classic problem and has been examined extensively in the literature. Our approach is applied to Dijkstra's algorithm.

Dijkstra's algorithm is a greedy algorithm. At each step, there exists a set of vertices S to which shortest paths are known, and a set T of vertices adjacent to members of this set. The best paths known to members of T are examined, and the vertex v, with the minimum path length is removed from T and added to S. A data structure that supports insert, delete, and deletemin can be used to implement this algorithm.

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
250,2500	0.15	0.14	0.06	33.33	2.50
500,5000	0.35	0.32	0.13	35.71	2.69
750,7500	0.56	0.52	0.19	36.61	2.95
1000,10000	0.79	0.73	0.25	37.97	3.16
2000,20000	1.74	1.65	0.52	37.64	3.35
2500,25000	2.22	2.08	0.65	38.51	3.42

Table 3: Shortest Path

Input graphs of $|V|$ vertices and $|E|$ edges were generated by choosing a set of $|E|$ distinct edges uniformly from all possible such sets, then rejecting graphs that were not connected. $|E|$ was chosen sufficiently large that each selection is connected with high probability, resulting in few rejections. The input sizes were chosen to keep the ration $|E|/|V|$ constant, for in practice the running time of the algorithm is affected by this ratio. Timing results are based on fifty executions at each input size. The size column of Table 3 contains an ordered pair indicating the number of vertices and edges.

6.4 Skyline

Given a set of rectangles with with collinear bottom edges, the *skyline* is the figure resulting from removing all hidden edges. The problem of computing the skyline of a set of rectangular buildings by eliminating hidden lines is discussed in [8]. The method used is divide and conquer and it constructs a skyline in $O(n\log(n))$ time. In this paper we use a plane sweep algorithm that can be easily implemented in terms of operations on priority queues. Plane sweep algorithms are widely used for computational geometry problems [9], and typically use a priority queue for event scheduling, and may be amenable to use of certification trail techniques.

Using a plane sweep algorithm, we compute the skyline as follows. Initialize a vertical sweep line to the left of all the rectangles (we may assume that all rectangle are to the right of the y-axis). As we sweep the line to the right we maintain a collection of the heights of the rectangles encountered. For each rectangle R, the height of R is added to the collection when we encounter R's left edge and removed when we encounter its right edge. The height of the skyline at any point x_0, is the maximum height in the collection when the sweepline is at $x = x_0$. Details are given below. A structure supporting insert and deletemin is

Size	Basic Algorithm	Generate Trail	Use Trail	% Saving	Speedup
1000	0.35	0.27	0.11	24.00	2.27
2000	0.66	0.59	0.22	27.66	2.55
5000	1.71	1.79	0.58	30.70	2.95
10000	3.86	4.01	1.17	32.90	3.30
20000	8.59	8.76	2.36	33.73	3.56
30000	13.20	14.02	3.55	33.90	3.74

Table 4: Skyline

all that is needed to order the events, and a structure supporting insert, max, and delete is required to store the rectangle heights. A priority queue (supporting insert and can be used to order the sweepline events, and a generalised priority queue to store the rectangle heights.

Input data was generated by choosing integral rectangle heights uniformly over the range $[0, 100000]$. The x-coordinates of the left edges were chosen uniformly over the range $[0, 90000]$ and the width of each rectangle was chosen uniformly over the range $[1, 10000]$. Timing results are based on twenty executions for each input size.

7 Conclusions

The experimental data in this paper shows the utility of the certification trail approach using abstract data types. This paper supplements [13] which provides experimental data illustrating the advantages of implementation specific certification trails over classical time redundancy. We have shown that the more general approach of checking abstract data types also provides performance superior to classical time redundancy. This is significant because a wide range of algorithms may be represented as a sequence of operations on abstract data types. The certification trail approach may therefore be used on these programs, without requiring per problem "ad hoc" techniques. Creation of library routines or class libraries for these data types allows the certification trail technique to be used transparently, and may allow it's use with only minor modifications of existing code.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Avisienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[3] Bayer, R., and McCreight, E., "Organisation of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[4] Chen, L., and Avisienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[5] Gabow, H. N., and Tarjan, R. E., "A linear-time algorithm for a special case of disjoint set union," *J. of Comp. and Sys. Sci.*, 30(2), pp. 209-221, 1985.

[6] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[7] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[8] Manber U., *Introduction to Algorithms: A Creative Approach* Addison-Wesley, Reading, MA, 1989.

[9] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

[10] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[11] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[12] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Digest of the 1991 Fault Tolerant Computing Symposium*, pp. 240-247, IEEE Computer Society Press, 1991.

[13] Sullivan, G.F., Wilson, D.S., Masson, G.M., Itoh, M., Smith, W.S., Kay, J.S., "Experimental evaluation of the certification trail method," Technical Report, Computer Science Department, The Johns Hopkins University

US005243607A

United States Patent [19]

Masson et al.

[11] Patent Number: 5,243,607

[45] Date of Patent: Sep. 7, 1993

[54] METHOD AND APPARATUS FOR FAULT TOLERANCE

[75] Inventors: Gerald M. Masson; Gregory F. Sullivan, both of Baltimore, Md.

[73] Assignee: The Johns Hopkins University, Baltimore, Md.

[21] Appl. No.: 543,451

[22] Filed: Jun. 25, 1990

[51] Int. Cl.[5] H04L 1/08

[52] U.S. Cl. 371/69.1; 371/68.3; 371/68.1; 371/19; 395/575

[58] Field of Search ..... 371/69.1, 68.3, 68.1, 371/19, 15.1, 16.1, 67.1; 364/200 MS File; 395/575

[56] References Cited

U.S. PATENT DOCUMENTS

4,696,003	9/1987	Kerr	371/69.1 X
4,756,005	7/1988	Shedd	371/69.1 X
5,005,174	4/1991	Bruckert et al.	371/68.3

OTHER PUBLICATIONS

H. Geng, "Circuit for the Complete Check of a Data-Processing System", IBM TDB, vol. 16, No. 4, Sep. 1974, pp. 1144-1145.

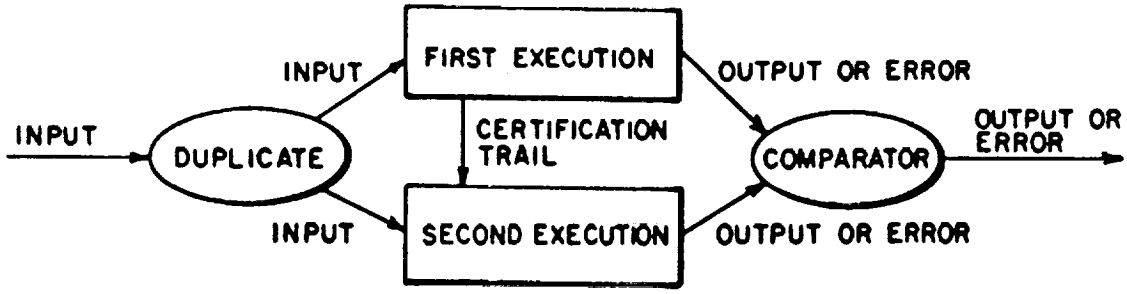
K. Knowlton, "A Combination Hardware-Software Debugging System," IEEE Transactions on Computers, Jan. 1968, pp. 81-86.

Primary Examiner—Robert W. Beausoliel, Jr.
Assistant Examiner—Ly V. Hua
Attorney, Agent, or Firm—Ansel M. Schwartz

[57] ABSTRACT

A method and apparatus for achieving fault tolerance in a computer system having at least a first central processing unit and a second central processing unit. The method comprises the steps of first executing a first algorithm in the first central processing unit on input which produces a first output as well as a certification trail. Next, executing a second algorithm in the second central processing unit on the input and on at least a portion of the certification trail which produces a second output. The second algorithm has a faster execution time than the first algorithm for a given input. Then, comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same. The step of executing a first algorithm and the step of executing a second algorithm preferably takes place over essentially the same time period.

18 Claims, 6 Drawing Sheets



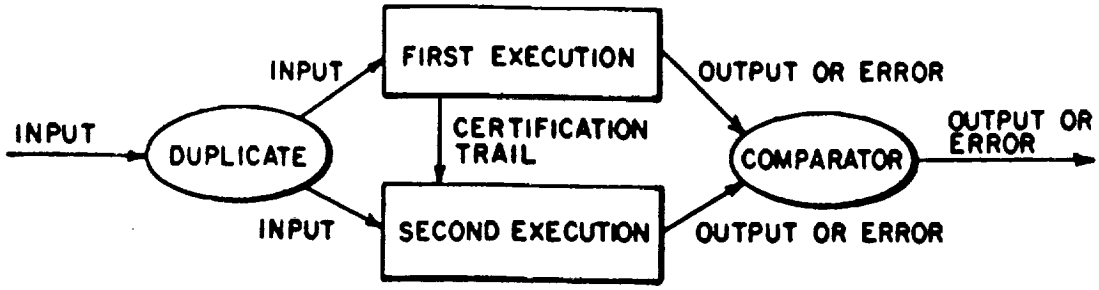


FIG. 1

```
Algorithm MINSPAN(G,weight)
Input: Connected graph G = (V,E) where V = {1,...,n} with edge weights.
Output: Spanning tree of G which has minimum weight
1  CHOOSE root ∈ V
2  FOR ALL u ∈ V, key(u) := ∞ END FOR
3  h := 0; v := root
4  WHILE v ≠ empty DO
5   key(v) := -∞
6   FOR EACH [v,w] ∈ E DO
7    IF weight([v,w]) < key(w) THEN
8     key(w) := weight([v,w]); prefer(w) := [v,w]
9     IF member(w,h) THEN changekey(w,key(w),h)
10     ELSE insert(w,key(w),h) END IF
11    END IF
12   END FOR
13   (v,k) := deletemin(h)
14 END WHILE
15 FOR ALL u ∈ V - {root}, OUTPUT(prefer(u)) END FOR
END MINSPAN
```

FIG. 3

FIG. 2(a)

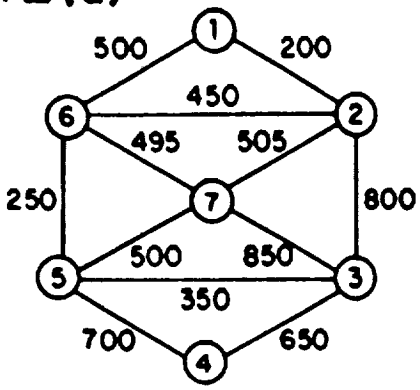


FIG. 2(b)

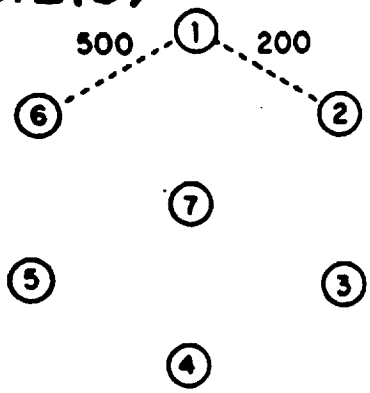


FIG. 2(c)

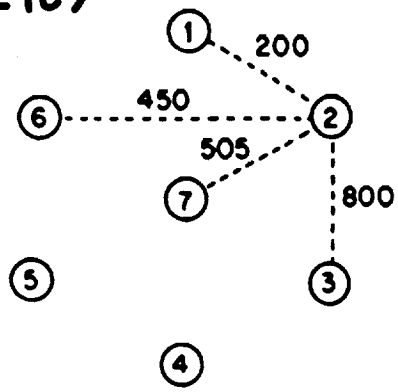


FIG. 2(d)

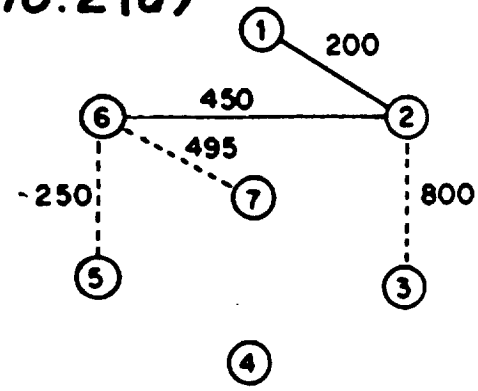


FIG. 2(e)

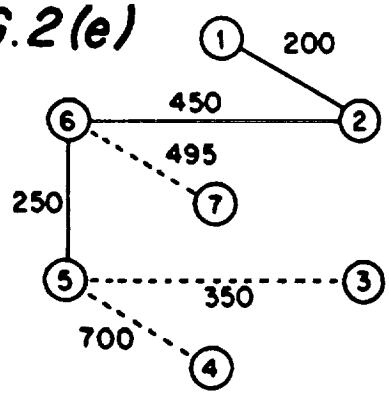
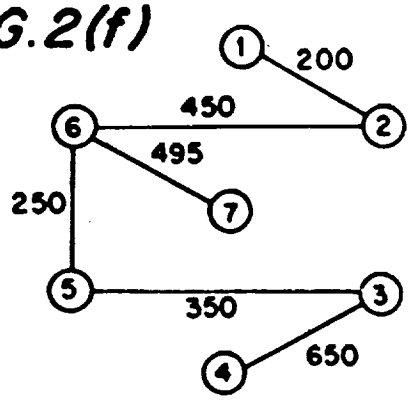


FIG. 2(f)



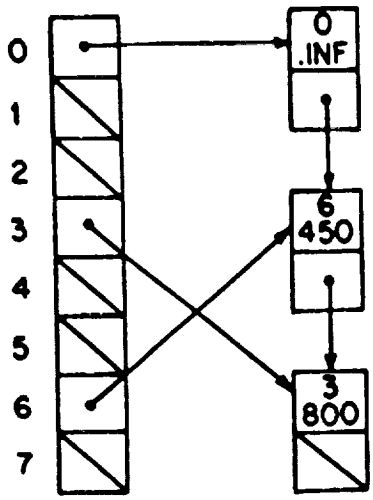


FIG. 4(a)

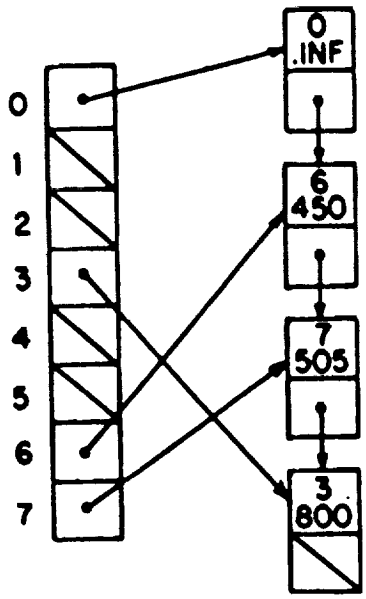


FIG. 4(b)

```
Algorithm HUFFMAN(FREQ)
Input: Sequence of positive integers FREQ.={f[1],f[2],...,f[n]}
Output: Pointer to a Huffman tree for the input frequencies
1  FOR i:=1 to n DO
2      insert (i,f[i],h)
3      ptr[i]:=allocate()
4      info[ptr[i]]:=(i,f[i])
5  END FOR
6  FOR j:=n+1 to 2n-1 DO
7      (item1, key1):=deletemin(h)
8      (item2, key2):=deletemin(h)
9      ptr[j]:=allocate()
10     info[ptr[j]]:=(j,key1+key2)
11     left[ptr[j]]:=ptr[item1]
12     right[ptr[j]]:=ptr[item2]
13     insert(j,key1+key2,h)
14 END FOR
15 OUTPUT(ptr[2n-1])
END HUFFMAN
```

FIG. 5

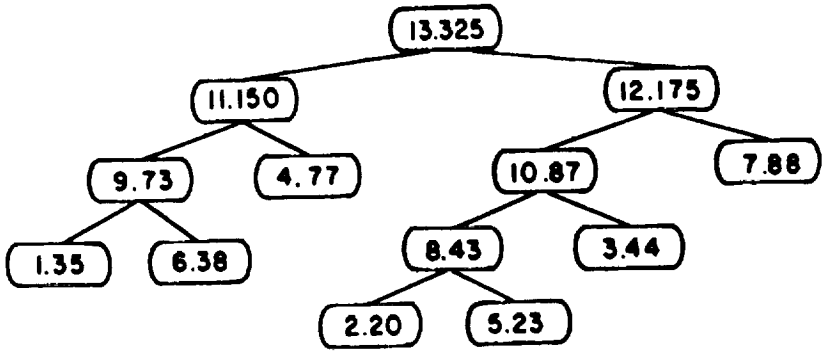


FIG. 6

```
Algorithm CONVEXHULL(S)
Input: Set of points, S, in R^2
Output: Counterclockwise sequence of points in R^2 which define convex hull of S
1 Let p1 be the point with the largest x coordinate (and smallest y to break ties)
2 For each point p (except p1) calculate the slope of the line through p1 and p
3 Sort the points (except p1) from the smallest slope to the largest. Call them p2,...pn
4 q1:=p1; q2:=p2; q3:=p3; m=3
5 FOR k = 4 to n DO
6   WHILE the angle formed by qm-1,qm, pk is ≥ 180 degrees DO m := m-1 END FOR
7   m := m+1
8   qm := pk
9 END FOR
10 FOR i = 1 to m DO, OUTPUT(qi) END FOR
END CONVEXHULL
```

FIG. 7

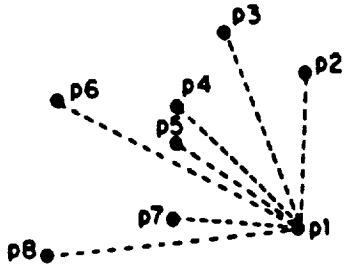


FIG. 8(a)

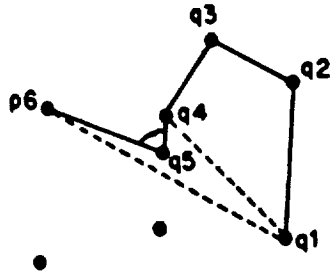


FIG. 8(b)

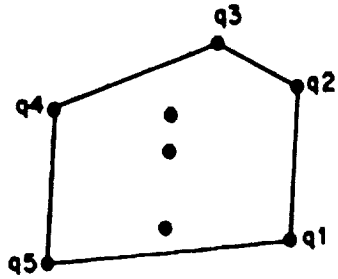


FIG. 8(c)

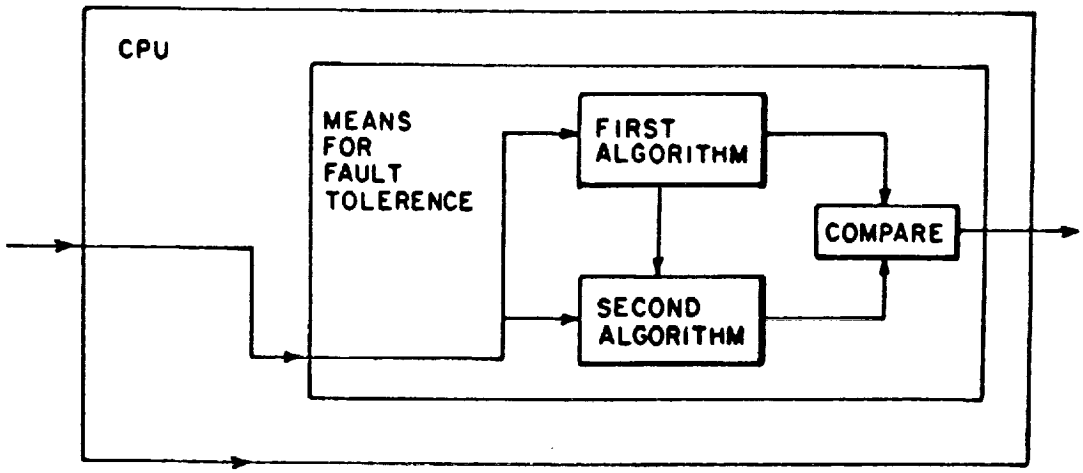


FIG. 9

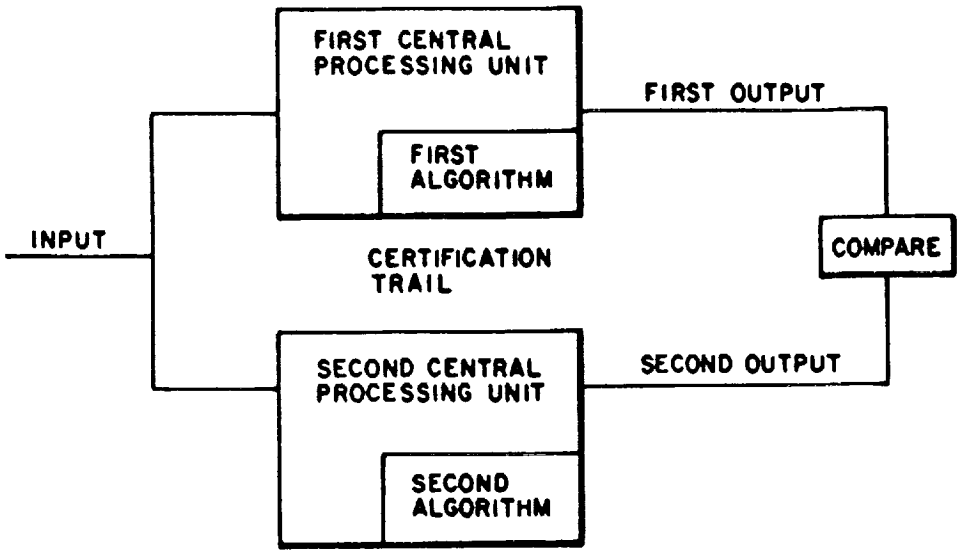

FIRST CENTRAL PROCESSING UNIT
FIRST ALGORITHM
FIRST OUTPUT
INPUT
CERTIFICATION TRAIL
COMPARE
SECOND CENTRAL PROCESSING UNIT
SECOND OUTPUT
SECOND ALGORITHM

FIG. 10

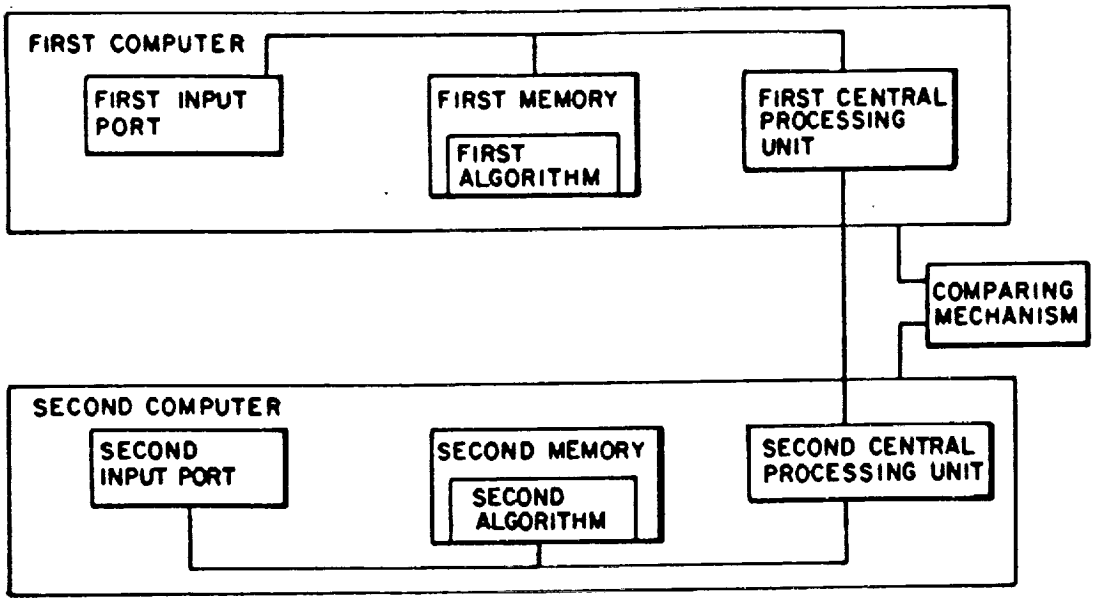

FIRST COMPUTER
FIRST INPUT PORT
FIRST MEMORY
FIRST ALGORITHM
FIRST CENTRAL PROCESSING UNIT
COMPARING MECHANISM
SECOND COMPUTER
SECOND INPUT PORT
SECOND MEMORY
SECOND ALGORITHM
SECOND CENTRAL PROCESSING UNIT

FIG. 11

METHOD AND APPARATUS FOR FAULT TOLERANCE

LICENSES

The United States Government has a paid-up non-exclusive license to practice the claimed invention herein as per NSF Grant CCR-8910569 and NASA Grant NSG 1442.

FIELD OF THE INVENTION

The present invention relates to fault tolerance. More specifically, the present invention relates to a first algorithm that provides a certification trail to a second algorithm for fault tolerance purposes.

BACKGROUND OF THE INVENTION

Traditionally, with respect to fault tolerance, the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so called time redundancy [Johnson, B., Design and analysis of fault tolerant digital systems, Addison-Wesley, Reading Mass., 1989; Siewiorek, D., and Swarz, R., The theory and practice of reliable design, Digital Press, Bedford, Mass., 1982]; however, it requires not additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, call N-version programming [Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," Digest of the 1978 Fault Tolerant Computing Symposium, pp. 3-9, IEEE Computer Society Press, 1978; Avizienis, A., "The N-version approach to fault tolerant software," IEEE Trans. on Software Engineering, vol. 11, pp. 1491-1501, December, 1985] (in this case $N=2$), allows for the detection of errors caused by some faults in the software in addition to those caused by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

SUMMARY OF THE INVENTION

The present invention pertains to a method for achieving fault tolerance in a computer system having at least a first central processing system and a second central processing system. The method comprises the steps of first executing a first algorithm in the first central processing unit on input which produces a first output as well as a certification trail. Next, executing a second algorithm in the second central processing unit on the input and on at least a portion of the certification trail which produces a second output. The second algorithm has a faster execution time than the first algorithm for a given input. Then, comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same. The step of executing a first algorithm and the step of executing a second algorithm preferably takes place over essentially the same time period.

The present invention also pertains to a method for achieving fault tolerance in a central processing unit. The method comprises the steps of executing a first algorithm in the central processing unit on input which produces the first output as well as a certification trail. Then, there is the step of executing a second algorithm in the central processing unit on the input and on at least a portion of the certification trail which produces a second output. The second algorithm has a faster execution time than the first algorithm for a given input. Then, there is the step of comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

The present invention also pertains to a computer system. The computer system comprises a first computer. The first computer has a first memory. The first computer also has a first central processing unit in communication with the memory. The first computer additionally has a first input port in communication with the memory in the first central processing unit. There is a first algorithm disposed in the first memory which produces a first output as well as a certification trail based on input received by the input port when it is executed by the first central processor. The computer system is additionally comprised of a second computer. The second computer is comprised of a second memory. The second computer is also comprised of a second central processing unit in communication with the memory and the first central processing unit. The second computer additionally is comprised of a second input port in communication with the memory in the second central processing unit. There is a second algorithm disposed in the second memory which produces a second output based on the input and on at least a portion of the certification trail when the second algorithm is executed by the second central processing unit. The second algorithm has a faster execution time than the first algorithm for a given input. The computer system is also comprised of a mechanism for comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

Moreover, the present invention also pertains to a computer. The computer is comprised of a memory. Additionally, the computer is comprised of a central processing unit in communication with the memory. The computer is additionally comprised of a first input port in communication with the memory and the central processing unit. There is a first algorithm disposed in the memory which produces a first output as well as a certification trail based on input received by the input port when the input is executed by the first central processor. There is a second algorithm also disposed in the memory which produces a second output based on the input and on at least a portion of the certification trail when the second algorithm is executed by the central processing unit. The second algorithm has a faster execution time than the first algorithm for a given input. Moreover, the computer is comprised of a mechanism for comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

BRIEF DESCRIPTION OF THE DRAWINGS

In the accompanying drawings, the preferred embodiments of the invention and preferred methods of practicing the invention are illustrated in which:

FIG. 1 is a block diagram of the present invention.

FIGS. 2A through FIG. 2F shows an examples of a minimum spanning tree algorithm.

FIG. 3 with the source code for a mince man algorithm.

FIG. 4A and 4B shows an example of a data structure used in the second execution of a mince man algorithm.

FIG. 5 with the source code for a Huffman algorithm.

FIG. 6 shows an example of a Huffman tree.

FIG. 7 with the source code for Graham's scan algorithm.

FIG. 8A through FIG. 8C shows a convex hull example.

FIG. 9 is a block diagram of an apparatus of the present invention.

FIG. 10 is a block diagram of another embodiment of the present invention.

FIG. 11 is a block diagram of another embodiment of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The central idea of the present invention, essentially a fault tolerance mechanism, as illustrated in FIG. 1, is to modify a first algorithm so that it leaves behind a trail of data which is called a certification trail. This data is chosen so that it can allow a second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. The outputs of the two executions are compared and are considered correct only if they agree. Note, however, care must be taken in defining this method or else its error detection capability might be reduced by the introduction of data dependent between the two algorithm executions. For example, suppose the first algorithm execution contains a error which causes an incorrect output and an incorrect trial of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions given below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the data trail. Finally, it should be noted that in FIG. 1 both executions can signal an error. These errors would include run-time errors such as divided-by-zero or nonterminating computation. In addition the second execution can signal error due to an incorrect certification trail. The fault tolerance means can be used in hardware or software systems and manifested as firmware or software in a central processing unit.

A formal definition of a certification trail is the following.

Definition 2.1. A problem P is formalized as a relation (that is, a set of ordered pairs). Let D be the domain (that is, the set of inputs) of the relation P and let S be the range (that is, the set of solutions) for the problem. It can be said an algorithm A solves a problem P if for all $d \in D$ when d is input to A then an $s \in S$ is output such that $(d,s) \in P$.

Definition 2.2. Let $P : D - S$ be a problem. Let T be the set of certification trails. A solution to this problem using a certification trail consists of two functions F_1 and F_2 with the following domains and ranges $F_1:D \rightarrow S \times T$ and $F_2:D \times T \rightarrow S \cup$ error. The functions must satisfy the following two properties:

(1) for all $d \in D$ there exists $s \in S$ and there exists $t \in T$ such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in P$

(2) for all $d \in D$ and for all $t \in T$ either ($F_2(d,t) = s$ and $(d,s) \in P$) or $F_2(d,t) =$ error.

The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct. It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

The certification trial approach also allows for the detection of faults in software. As in N-version programming, separate teams can write the specification now must include precise information describing the generation and use of the certification trial. Because of the additional data available to the second execution, the specifications of the two phases can be very different; similarly, the two algorithms used to implement the phases can be very different. This will be illustrated in the convex hull example to be considered later. Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. This will be illustrated in the minimum spanning tree and Huffman tree examples to be considered later. When significantly different algorithms are used it is sometimes possible to save programming effort by sharing program code. While this reduces the ability to detect errors in the software it does not change the ability to detect transient hardware errors as discussed earlier.

With respect to the above, it has been assumed that our method is implemented with software; however, it is clearly possible to implement the certification trail technique by using dedicated hardware. It is also possible to generalize the basic two-level hierarchy of the certification trial approach as illustrated in FIG. 1 to higher levels.

Examples of the Certification Trail Technique

In this section, there is illustrated the use of certification trails by means of applications to three well-known and significant problems in computer science: the minimum spanning tree problem, the Huffman tree problem, and the convex hull problem. It should be stressed here that the certification trail approach is not limited to these problems. Rather, these algorithms have been selected only to give illustrations of this technique.

Minimum Spanning Tree Example

The minimum spanning tree problem has been examined extensively in the literature and an historical survey is given in [Graham, R.L., "An efficient algorithm for determining the convex hull of a planar set", Information Processing Letters, pp. 132-133, 1, 1972]. The certification trial approach is applied to a variant of the Prim/Dijkstra algorithm]Prim, R.C., "Shortest connection networks and some generalizations,: Bell Syst. Tech. J., pp. 1389-1401, November, 1957; Dijkstra, E.

W., "A note on two problems in connexion with graphs," Numer. Math. 1, pp. 269-1984, Jun. 20-22] as explicated in [Tarjan, R.E., Data Structures and Network Algorithms, Society for Industrial and applied Mathematics, Philadelphia, Pa. 1983]. The discussion of the application of the certification trail approach to the minimum spanning tree problem beings with some preliminary definitions.

Definition 3.1. A graph G = (V,E) consists of a vertex set V and an edge set E. An edge is an unordered pair of distinct vertices which is notated as, for example, [v,w], and it is said v is adjacent to w. A path in a graph from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $[v_1, v_{1=1}]$ is an edge for $i \in [1, \ldots, k-1]$. A path is a cycle if $k > 1$ and $v_1 = v_k$. An acyclic graph is a graph which contains no cycles. A connected graph is a graph such that for all pairs of vertices v,w there is a path from v to w. A tree is an acyclic and connected graph.

Definition 3.2. Let G = (V,E) be a graph and let w be a positive rational valued function defined on E. A subtree of G is a tree, T(V',E'), with $V' \subseteq V$ and $E' \subseteq E$. It is said T spans V' and V' is spanned by T. If V' = V then we say T is a spanning tree of G. The weight of this tree is $\Sigma_{e \in E} w(e)$. A minimum spanning tree is a spanning tree of minimum weight.

Data Structures and Supported Operations

Before discussion of the minimum spanning tree algorithm, there must be described the properties of the principle data structure that are required. Since many different data structures can be used to implement the algorithm, initially there is described abstractly the data that can be stored by the data structure and the operations that can be used to manipulate this data. The data consists of set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set; however, at all times, the item numbers of distinct ordered pairs must be distinct. It is possible, through, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, k is a key value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: (i,k) < (i',k') iff k < k' or (k = k' and i < i'). The data structure should support a subset of the following operations.

member (i,h) returns a boolean value of true if h contains an ordered pair with item number i, otherwise returns false.

inser: (i,k,h) adds the ordered pair (i,k) to the set h.

delete (i,h) deletes the unique ordered pair with item number i from h.

changekey (i,k,h) is executed only when there is an ordered pair with item number i and h. This pair is replaced by (i,k).

deletemin (h) returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If h is the empty set then the token "empty" is returned.

predecessor (i,h) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned.

Many different types and combinations of data structures can be used to support these operations efficiently. In our case, there is used two different data structure methods to support these operations. One method will be used in the first execution of the algorithm and another, faster and simpler, method will be used in the second execution. The second method relies on a trail of data which is output by the first execution.

MINSPAN ALGORITHM

Before discussing precise implementation details for these methods the overall algorithm used in both executions is presented. Pidgin code for this algorithm appears below. In addition, FIG. 2 illustrates the execution of the algorithm on a sample graph and the table below records the data structure operations the algorithm must perform when run on the sample graph. The fist column of the table gives the operations except member and the parameter h dropped to reduce clutter. The second column gives the evolving contents of h. The third column records the ordered pair deleted by the deletemin operation. The fourth column records to certification trail corresponding to these operations and is further discussed below.

The algorithm uses a "greedy" method to "grow" a minimum spanning tree. The algorithm starts by choosing an arbitrary vertex from which to grow the tree. During each iteration of the algorithm a new edge is added to the tree being constructed. Thus, the set of vertices spanned by the tree increases by exactly one vertex for each iteration. The edge which is added to the tree is the one with the smallest weight. FIG. 2 shows this process in action. FIG. 2(*a*) shows the input graph, FIGS. 2(*b*) through 2(*e*) show several stages of the tree growth and FIG. 2(*f*) shows the final output of the minimum spanning tree. The solid edges in FIGS. 2(*b*) through 2(*e*) represent the current tree and the dotted edges represent candidates for addition to the tree.

To efficiently find the edge to add to the current tree the algorithm uses the data structure operations described above. As soon as a vertex, say v, is adjacent to some vertex which is currently spanned it is inserted in the set h. The key value for v is the weight of the minimum edge between v and some vertex spanned by the current tree. The array element prefer (v) is used to keep track of this minimum weight edge. As the tree grows, information is updated by operations such as insert (i,k,h) and changekey (i,k,h).

TABLE 1

Data structure operations and certification trail for MINSPAN

Operation	Set of Ordered Pairs	Delete	Trail
insert(2,200)	(2,200)		smallest
insert(6,500)	(2,200),(6,500)		2
deletemin	(6,500)	(2,200)	
insert(3,800)	(6,500),(3,800)		6
changekey(6,450)	(6,450),(3,800)		smallest
insert(7,505)	(6,450),(7,505),(3,800)		6
deletemin	(7,505),(3,800)	(6,450)	
insert(5,250)	(5,250),(7,505),(3,800)		smallest
changekey(7,495)	(5,250),(7,495),(3,800)		5
deletemin	(7,495),(3,800)	(5,250)	
changekey(3,350)	(3,350),(7,495)		smallest
insert(4,700)	(3,350),(7,495),(4,700)		7
deletemin	(7,495),(4,700)	(3,350)	
changekey(4,650)	(7,495),(4,650)		7
deletemin	(4,650)	(7,495)	
deletemin		(4,650)	
deletemin		empty	

The deletemin (h) operation is used to select the next vertex to add to the span of the current tree. Note, the algorithm does not explicitly keep a set of edges representing the current tree. Implicitly, however, if (v,k) is returned by deletemin then prefer (v) is added to the current tree.

In the first execution of the MINSPAN algorithm, the MINSPAN code is used and the principle data structure is implemented with a balanced tree such as an AVL tree [Adel'son-Vel'skii, G.M., and Landis, E.M., "An algorithm for the organization of information", Soviet Math. Dokl., pp. 1259-1262, 3, 1962], a red-black tree [Guibas, L.J., and Sedgewick, R., "A dichromatic Framework for balanced trees", Proceedings of the Nineteenth Annual Symposium on Foundations of Computing, pp. 8-21, IEEE Computer Society Press, 1978] or a b-tree [Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972]. In addition, an array of pointers indexed from 1 to n is used. The balanced search tree stores the ordered pairs in h and is based on the total order described earlier. The array of pointers is initially all nil. For each item i, the ith pointer of the array is used to point to the location of the ordered pair with item number i in the balanced search tree. If there is no such ordered pair in the tree then the ith pointer is nil. This array allows rapid execution of operations such as member (i,h) and delete (i,h).

The certification trail is generated during the first execution as follows: When CHOOSE root ϵ V is executed in the first step, the vertex which is chosen is output. Also, each time insert (i,k,h) or changekey (i,k,h) are executed, predecessor (i,h) is executed afterwards, and the answer returned is output. This is illustrated in column labeled "Trail" in the table above.

The second execution of the MINSPAN algorithm also uses the MINSPAN code; however, the CHOOSE construct and the data structure operations are implemented differently than in the fist execution. The CHOOSE is performed by simply reading the first element of the certification trail. This guarantees the same choice of a starting vertex is made in both executions. FIG. 4 depicts the principal data structure used which is called an indexed linked list. The array is indexed from 1 to n and contains pointers to a singly linked list which represents the current contents of h from smallest to largest. The ith element of the array points to the node containing the ordered pair with the item number i if it is present in h; otherwise, the pointer is nil. The 0th element of the array points to the node containing (0, -INF). Initially, the array contains nil pointers except the 0th element. In order to implement the data structure operations, the following is provided.

To perform insert (i,k,h), it is necessary to read the next value in the certification trail. This value, say j, is the item number of the ordered pair which is the predecessor of (i,k) in the current contents of h. A new linked list node is allocated and the trail information is used to insert the node into the data structure. Specifically, the ith array pointer is traversed to a node in the linked list, say Y. (If j = "smallest" then the 0th array pointer is traversed.) The new node is inserted in the list just after node Y and before the next node in the linked list (if there is one). The data field in the new node is set to (i,k) and the ith pointer of the array is set to point to the new node. FIG. 4 shows the insertion of (7,505) into the data structure given that the certification trail value is 6. FIG. 3(*a*) is before the insertion and FIG. 3(*b*) is after the insertion.

When the insert operation is performed, some checks must be conducted. First, the ith array pointer must be nil before the operation is performed. Section, the sorted order of the pairs stored in the linked list must be preserved after the operation. That is, if (i',k') is stored in the node before (i,k) in the linked list and (i'',k'') is stored after (i,k), then (i',k') < (i,k) < (i'', k'') must hold in the total order. If either of these checks fails then execution halts and "error" is output.

To perform delete (i,h) the ith array pointer is traversed and the node found is deleted from the linked list. Next, the ith array pointer is set to nil. FIG. 4 shows the deletion of item number 7 if one considers FIG. 3(*a*) as depicting the data structure before the operation and FIG. 3(*b*) depicting it afterwards. When the delete operation is performed one check is made. If the ith array pointer is nil before the operation then the execution halts and "error" is output.

To perform changekey (i,i,h) it suffices to perform delete (i,h) followed by insert (i,k,h). Note, this means the next item in the certification trail is read. Also, the checks associated with both these two operations are performed and the execution halts with "error" output if any check fails.

To perform detelemin (h) the 0th array pointer is traversed To the head of the list and the next node in the list is accessed. If there is no such node then "empty" is returned and the operation is complete. Otherwise, suppose the node is Y and suppose it contains the ordered pair (i,k), then the node Y is deleted from the list, the ith array pointer is set to nil, and (i,k) is returned.

Lastly, to perform member (i,h) the ith array pointer is examined. If it is nil then false is returned, otherwise, true is returned. The predecessor (i,h) operation is not used int he second execution.

This completes the description of the second execution. To show that there is described a correct implementation of the certification trail method requires a proof. The proof has several parts of varying difficulty. First, one must show that if the first execution is fault-free then it outputs a minimum spanning tree. Second, one must show that if the first and second executions are fault-free then they both output the same minimum spanning tree. Both these parts of the proof are not difficult to show.

The third more subtle part of the proof deals with the situation in which only the second execution is fault-free. This means an incorrect certification trail may be generated in the first execution. In this case, it must be shown that the second execution outputs either the correct minimum spanning tree or "error". The checks that were described this property by detecting any errors that would prevent the execution from generating the correct output.

In the first execution each data structure operation can be performed in O(log(n)) time where [V]=n. There are at most O(m) such operations and O(m) additional time overhead where [E]=m. Thus, the first execution can be performed in O(mlog(n)). It is noted that th is algorithm does not achieve the fastest known asymptotic time complexity which appears in Gabow, H.N., Galil, Z., Spencer, T., and Tarjan, R.E., "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," Combinatorica 6, pp. 109-122, 2, 1986. However, the algorithm presented here has a significantly smaller constant of proportion-

ally which makes it competitive for reasonably sized graphs. In addition, it provides us with a relatively simple and illustrative example of the use of a certification trail.

In the second execution each data structure operation can be performed in O(1). There are still at most O(m) such operations and O(m) additional time overhead. Hence, the second execution can be performed in O(m) time. In other words, because of the availability of the certification trail, the second execution is performed in linear time. There are no known O(m) time algorithms for the minimum spanning tree problem. Komlos [26] was able to show that O(m) comparisons suffice to find the minimum spanning tree. However, there is no known O(m) time algorithm to actually find and perform these comparisons. Even the related "verification problem has no known linear time solution. In the verification problem the input consists of an edge weighted graph and a subtree. The output is "yes" if the subtree is the minimum spanning tree and "no" otherwise. The best known algorithm for this problem was created by Tarjan [Tarjan, R.E., "Applications of path compression on balanced trees", J. ACM, pp. 690–715, October, 1979] and has the nonlinear time complexity of O(-mα(m,n)), where α(m,n) is a functional inverse of Ackerman's function. The fact that the data in a certification trail enables a minimum spanning tree to be found in linear time is, we believe, intriguing, significant, and indicative of the great promise of the certification trail technique.

Huffman Tree Example

Huffman trees represent another classic algorithmic problem, one of the original solutions being attributed to Huffman [Huffman, D., "A method for the construction of minimum redundancy codes", Proc. IRE, pp. 1098–1101, 40, 1952]. This solution has been used extensively to perform data compression through the design and use of so-called Huffman codes. These codes are prefix codes which are based on the Huffman tree and which yield excellent data compression ratios. The tree structure and the code design are based on the frequencies of individual characters in the data to be compressed. See Huffman, D., "A method for the construction of minimum redundancy codes", Proc. IRE, pp. 1098–1101, 40, 1952, for information about the coding application.

Definition 3.3. The Huffman tree problem is the following: Given a sequence of frequencies (positive integers) f[1], f[2], . . . , f[n], construct a tree with n leaves and with one frequency value assigned to each leaf so that the weighted path length is minimized. Specifically, the tree should minimize the following sum: $\Sigma_{l_i \in LEAF} \text{len}(i) f[i]$ where LEAF is the set of leaves, len(i) is the length of the path from the root of the tree to the leaf l_i, f[i] is the frequency assigned to the leaf l_i.

An example of a Huffman tree is given in FIG. 6. The input frequencies are: f(1) = 35, f(2) = 20, f(3) = 44, f(4) = 77, f(5) = 23, f(6) = 38, and f(7) = 88. The frequencies appear inside the leaf nodes as the second elements of the ordered pairs in the figure.

HUFFMAN ALGORITHM

The algorithm to construct the Huffman tree uses a data structure which is able to implement the insert and the deletemin operations which are defined above in the minimum spanning tree example. This type of data structure is often called a priority queue. The algorithm also uses the command allocate to construct the tree. This command allocates a new node and returns a pointer to it. Each node is able to store an item number and a key value in the field called info. the item numbers are in the set (1, . . . , 2n − 1) and the key values are sums of frequency values. The nodes also contain fields for left and right pointers since the tree being constructed is binary.

The Huffman tree is built from the bottom up and the overall structure of the algorithm is based on the greedy "merging" of subtrees. An array of pointers called ptr is used to point to the subtrees as they are constructed. Initially, n single vertex subtrees with the smallest associated frequency values. To perform a merge a new subtree is created by first allocating a new root node and next setting the left and right pointers to the two subtrees being merged. The frequency associated with the new subtree is the sum of the frequencies of the two subtrees being merged. In FIG. 6 the frequency associated with each subtree is shown as the second value in the root vertex of the subtree. Details of the algorithm are given below. Note that the priority queue data structure allows the algorithm to quickly determine which subtrees should be merged by enabling the two smallest frequency values to be found efficiently during each iteration.

Table 2 below illustrates the data structure operations performed when the Huffman tree in FIG. 6 is constructed. For conciseness the initial n inset operations have been omitted. The first column gives the set of ordered pairs in h. The second column gives the result of the two deletemin operations during each iteration. Note that this column is labeled "Trail" because it is also output as the certification trail. The third column records the elements which are inserted by the command on line 13.

TABLE 2

Data structure operations and certifications trial for HUFFMAN

Set of Ordered Pairs	Trail	Insert
(2,20),(5,23),(1,35),(6,38),(3,44),(4,77),(7,88)		
(1,35),(6,38),(8,43),(3,44),(4,77),(7,88)	(2,20),(5,23)	(8,43)
(8,43),(3,44),(9,73),(4,77),(7,88)	(1,35),(6,38)	(9,73)
(9,73),(4,77),(10,87),(7,88)	(8,43),(3,44)	(10,87)
(10,87),(7,88),(11,150)	(9,73),(4,77)	(11,150)
(11,150),(12,175)	(10,87),(7,88)	(12,175)
(13,325)	(11,150),(12,175)	(13,325)

First Execution of HUFFMAN

In this execution the code entitled HUFFMAN is used and the priority queue data structure is implemented with a heap [Tarjan, R.E., Data Structures and Network Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, Pa. 1983] or a balanced search tree [Guibas, L.J., and Sedgewick, R., "A dichromatic framework for balanced trees", Proceedings of the Nineteenth Annual Symposium on Foundations of Computing, pp. 8–21, IEEE computer Society Press, 1978; Adel'son-Vel-Vel'skii, G.M., and Landis, E.M., "An algorithm for the organization of information", Soviet Math. Dokl., pp. 1259–1262, 3, 1962; Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp. 173–189, 1, 1972]. Actually, any correct implementation is acceptable; however, to achieve a reasonable time complexity for this execution the suggested implementation are desir-

C-3.

able. the certification trail is generated as follows: whenever deletemin (h) is executed the item number and the key value which are returned are both output. In the table, the certification trail is listed in the second column.

Second Execution of HUFFMAN

This execution consists of two parts which may be logically separated but which are performed together. In the first logical part, the code called HUFFMAN is executed again except that the data structure operations are treated differently. All insert operations are not performed and all deletemin operations are performed by simply reading the ordered pairs from the certification trail. In the second logical part, the data structure operations are "verified". Note, by "verify" it does not mean a formal proof of correctness based on the text of an algorithm. The problem of verification can be formulated as follows: given a sequence of insert (i,k,h) and deletemin (h) operations (h) operations check to see if the answers are correct. It should be noted that while in our example there is only one h, in general there can be multiple h's to be handled.

The description of the algorithm for the second execution can be further simplified because only some restricted types of operation sequences are generated by the HUFFMAN code. First, it can be observed that all elements are ultimately deleted from h before the algorithm terminates; second, it can be further observed that when an element is inserted into h, its key value is larger than the key value of the last element deleted from h. These two important observations allow us to check a sequence using the simplified method which is described next.

Our simplified method uses an array of integers indexed from 1 to $2n-1$. This array is used to track the contents of h. If the ordered pair (i,k) is in h, then array element i is set to a value of k; and if no ordered pair with item number i is in h, then array element i is set to a value of -1. Initially, all array elements are set to -1 and then operation sequence is processed. If insert (i,k) is executed then array element i is checked to see if it contains -1. (The value of -1 is an arbitrary selection meant to serve only as an indicator.) If array element i does contain -1, then it is set to k. If deletemin (h) is executed, then the answer indicated by the certification trail, say (i,k), is examined. Array element i is checked to see if it contains k. In addition, k is compared to the key value of previous element in the certification trail sequence to see if it is greater than or equal to that previous value. If both these checks succeed then array element i is set to -1.

If any of the checks just described above fails, then the execution halts and "error" is output. Otherwise the operation sequence is considered "verified". It can be rigorously shown that the checks described are sufficient for determining whether the answers given in the certification trail are correct; this proof, however, has been omitted for the sake of brevity. Finally, it is worth noting that to combine the two logical parts of this execution, one can perform the data structure checking in tandem with the code execution of HUFFMAN. Each time an insert or deletemin is encountered in the code, the appropriate set of checks are performed.

Time Complexity Comparison of the Two Executions

Again, as in the minimum spanning tree example, the availability of the certification trail permits the second execution for the Huffman tree problem to be dramatically more efficient than the first.

In the first execution of HUFFMAN, each data structure operation can be performed in O(log(n)) time where n is the number of frequencies in the input. There are O(n) such operations and O(n) additional time overhead, hence, the execution can be performed in O(n log (n)). This is the same complexity as the best known algorithm for constructing Huffman trees.

In the second code execution of HUFFMAN, each data structure operations is performed in constant time. Further, verifying the data structure operations are correct takes only a constant time per operation. Thus, it follows that the overall complexity of the second execution is only O(n).

Convex Hull Example

The convex hull problem is fundamental in computational geometry. The certification trail solution to the generation of a convex hull is based on a solution due to Graham [Graham, R.L., "An efficient algorithm for determining the convex hull of a planar set", Information Processing Letters, pp. 132-133, 1 1972] which is called "Graham's Scan." (For basic definitions and concepts in computational geometry, see the text of Preparata and Shamos [Preparata F.P., and Shamos M.I., Computational geometry; an introduction, Springer-Verlag, New York, N.Y., 1985].) For simplicity in the discussion which follows, it is assumed the points are in so-called "general position" (this is, no three points are colinear). It is not difficult to remove this restriction.

Definition 3.4. A convex region in R^2 is a set of points, say Q, in R^2 such that for every pair of points in Q the line segment connecting the points lies entirely within Q. A polygon is a circularly ordered set of line segments such that each line segment shares one of its endpoints with the preceding line segment and shares the other endpoint with the succeeding line segment in the ordering. The shared endpoints are called the vertices of the polygon. A polygon may also be specified by an ordering of its vertices. A convex polygon is a polygon which is the boundary of some convex region. The convex hull of a set of points, S, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique and its vertices are a subset of the points in S. It is specified by a counterclockwise sequence of its vertices.

FIG. 8(c) shows a convex hull for the points indicated by black dots. Graham's can algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects an "extreme" point and calls it p_1. The next two steps sort the remaining points in a way which is depicted in FIG. 8(a). It is not hard to show that after these three steps the points when taken in order, $P_1, p_2, \ldots, p_n$, form a simple polygon; although, in general, this polygon is not convex.

Graham's Scan Algorithm

It is possible to think of Graham's scan algorithm as removing points from this simple polygon until it becomes convex. the main FOR loop iteration adds vertices to the polygon under construction and the inner WHILE loop removes vertices from the construction. A point is removed when the angle test performed at

Step 6 reveals that it is not on the convex hull because it falls within the triangle defined by three other points. A "snapshot" of the algorithm given in FIG. 8(*b*) shows that q_5 is removed from the hull. The angle formed by q_4,q_5, p_6 is less than 180 degrees. This means, q_5 lies within the triangle formed by q_4, p_1, p_6. (Note, $q_1 = p_1$.) In general, when the angle test is performed, if the angle formed by qm − 1,qm,pk is less than 180 degrees, then qm lies within the triangle formed by qm − 1,p1,pk. Below it will be revealed that this is the primary information relied on in our certification trail. When the main FOR loop is complete, the convex hull has been constructed.

First Execution of Graham's Scan

In this execution the code CONVEXHULL is used. The certification trail is generated by adding an output statement within the WHILE loop. Specifically, if an angle of less than 180 degrees is found in the WHILE loop test then the four tuple consisting of qm,qm − 1,p1,pk is output to the certification trail. Table 3 below shows the four tuples of points that would be output by the algorithm when run on the example in FIG. 8. The points in Table 3 are given the same names as in FIG. 8(*a*). The final convex hull points q1, . . . qm are also output to the certification trail. Strictly speaking the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $s_1,s_2, \ldots, s_n$ then rather than output the element in R^2 corresponding to s_i the number i is output. It is not hard to code the program so that this is done.

TABLE 3

First part of certification trail for Graham's scan

Point not on convex hull	Three surrounding points
p_5	p_4-p_1-p_6
p_4	p_3-p_1-p_6
p_7	p_6-p_1-p_8

Second Execution for the Convex Hall Problem

Let the certification trail consist of a set of four tuples, (x_1,a_1,b_1,c_1), (x_2,a_2,b_2,c_2), . . . , (x_r,a_r,b_r,c_r) followed by the supposed convex hull, $q_1,q_2, \ldots ,q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

First, the algorithm checks for $i \in (1, \ldots ,r)$ that x_i lies within the triangle defined by a_i,b_i and c_i.

Second, the algorithm checks that for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is less than or equal to 180 degrees.

Third, it checks that there is a one to one correspondence between the input points and the points in $(x_1, \ldots ,x_r) \cup (q_1, \ldots ,q_m)$.

Fourth, it checks that for $i \in (1, \ldots ,r)$, a_i,b_i, and c_i are among the input points.

Fifth, it checks that there is a unique point among the points on the supposed convex hull which is a local extreme point. A point q on the hull is a local extreme point if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. As mentioned above, the trail data actually consists of indices into the input data. this does not unduly complicate the checks above; instead it makes them easier. The correctness and adequacy of these checks must be proven.

Time Complexity of the Two Executions

In the first execution the sorting of the input points takes O(nlog(n)) time where n is the number of input points. One can show that this cost dominates and the overall complexity is O(nlog(n)).

It is possible to note that, unlike the minimum spanning tree example and the Huffman tree example, the convex hull example utilizes an algorithm in the second execution that is not a close variant of that used int he first execution. However, like the previous two examples, the second execution for the convex hull problem depends fundamentally on the information in the certification trail for efficiency and performance.

Concurrency of Executions

In the three examples discussed above, it is possible to start the second execution before the first execution has terminated. This is a highly desirable capability when additional hardware is available to run the second execution (for example, with multiprocessor machines, or machines with coprocessors or hardware monitors).

In the case of the minimum spanning tree problem, the two executions can be run concurrently. It is only necessary for the second execution to read the certification trail as it is generated—one item number at a time. Thus, there is a slight time lag in the second execution. The case of the Huffman tree problem is similar. Both executions can be run concurrently if the second execution reads the certification trail as it is generated by the first execution.

The case of the convex hull problem is not quite as favorable, but it is still possible to partially overlap the two executions. For example, as each 4-tuple of points is generated by the first execution, it can be checked by the second execution. But the second execution must wait for the points on the convex hull to be output at the end of the first execution before they can be checked.

An additional opportunity for overlapping execution occurs when the system has a dedicated comparator. In this case it is sometimes possible for the two executions to send their output to the comparator as they generate it. For example, this can be done in the minimum spanning tree problem where the edges of the tree can be sent individually as they are discovered by both executions.

Comparison of Techniques

The certification trail approach to fault tolerance, whether implemented in hardware or software or some combination thereof, has resemblances with other fault tolerant techniques that have been previously proposed and examined, but in each case there are significant and fundamental distinctions. These distinctions are primarily related to the generation and character of the certification trail and the manner in which the secondary algorithm or system uses the certification trail to indicate whether the execution of the primary system or algorithm was in error and/or to produce an output to be compared with that of the primary system.

To being, the certification trail approach might be viewed as a form of N-version programming [Chen, L., and Avizienis A., "N-version programming: a fault

tolerant approach to reliability of software operation," Digest of the 1978 Fault Tolerant Computing Symposium, pp. 3-9, IEEE computer Society Press, 1978; Avizienis, A., and Kelly J., "Fault tolerance by design diversity: concepts and experiments," Computer, vol. 17, pp. 67-80. August, 1984]. This approach specifies that N different implementations of an algorithm be independently executed with subsequent comparison of the resulting N outputs. There is no relationship among the executions of the different versions of the algorithms other than they all use the same input; each algorithm is executed independently without any information about the execution of the other algorithms. In marked contrast, the certification trail approach allows the primary system to generate a trail of information while executing its algorithm that is critical to the secondary system's execution of its algorithm. In effect, N-version programming can be thought of relative to the certification trail approach as the employment of a null trail.

A software/hardware fault tolerance technique known as the recovery block approach [Randell, Ba., "System structure for software fault tolerance," IEEE Trans. on Software Engineering vol. 1, pp. 202-232, June, 1975; Anderson, T., and Lee, P., Fault tolerance: principles and practices, Prentice-Hall, Englewood Cliffs, N.J., 1981; Lee, Y. H. and Shin, K. G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," IEEE Trans. Comput., vol C-33, pp. 113-124, February 1984.] uses acceptance tests and alternative procedures to produce what is to be regarded as a correct output from a program. When using recovery blocks, a program is viewed as being structured into blocks of operations which after execution yield outputs which can be tested in some informal sense for correctness. The rigor, completeness, and nature of the acceptance test is left to the program designer, and many of the acceptance tests that have been proposed for use tend to be somewhat straightforward [Anderson, T., and Lee, P., Fault tolerance: principles and practices, Prentice-Hall, Englewood Cliffs, N.J., 1981]. Indeed, formal methodologies for the definition and generation of acceptance tests have thus far not been established. Regardless, the certification trail notion of a secondary system that receives the same input as the primary system and executes an algorithm that takes advantage of this trail to efficiently produce the correct output and/or to indicate that the execution of the first algorithm was correct does not fall into the category of an acceptance test.

A watchdog processor is a small and simple (relative to the primary system being monitored) hardware monitor that detects errors examining information relative to the behavior of the primary system [Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors," IEEE Trans. on Computers, vol. 37, pp. 160-174, February, 1988; Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors—a survey," IEEE Trans. on Computers, vol. 37, pp. 160-174, February, 1988; Namjoo, M., and McCluskey, E., "Watchdog processors and capability checking," Digest of the 1982 Fault Tolerant Computing Symposium, pp. 245-248, IEEE Computer Society Press, 1982.]. Error detection using a watchdog processor is a two-phase process: in the set-up phase, information about system behavior is provided a priori to the watchdog processor about the system to be monitored; in the monitoring phase, the watchdog processor collects or is sent information about the operation of the system to be compared with that which was provided during the set-up phase. On the basis of this comparison, a decision is made by the watchdog processor as to whether or not an error has occurred. The information about system behavior by means of which a watchdog processor must monitor for errors includes memory access behavior [Namjoo, M., and McCluskey, E., "Watchdog processors and capability checking," Digest of the 1982 Fault Tolerant Computing Symposium, pp. 245-248, IEEE Computer Society Press, 1982], control and program flow [Eifert, J. B. and Shen, J. P., "Processor monitoring using asynchronous signatured instruction streams," Dig. 14th Int. Conf. Fault-Tolerant Comput., pp. 394-399, 1984, June 20-22; Iyengar, V. S. and Kinney, L. L., "Concurrent fault detection in microprogrammed control units," IEEE Trans. Comput., vol. C-34, pp. 810-821, September 1985; Kane, J. R. and Yau, S. S., "Concurrent software fault detection, " IEEE Trans. Software Eng., vol. SE-1, pp. 87-99, March 1975; Lu, D., "Watchdog processor and structural integrity checking, " IEEE Trans. Comput., vol. C-31, pp. 681-685, July 1982; Namjoo, M., "Techniques for concurrent testing of VLSI processor operation," Dig. 1982 Int. Test Conf., pp. 461-468, November 1982; Namjoo, M., "CERBERUS-16: An architecture for a general purpose watchdog processor," Dig. Papers 13th Annu. Int. Sump. Fault Tolerant Comput., pp. 216-219, June, 1983; Shen, J. P. and Schuette, M.A., "On-line self-monitoring using signatured instruction streams," Proc. 1983 Int. Test Conf., pp. 275-282, October, 1983; Sridhar, T. and Thatte, S. M., "Concurrent checking of program flow in VLSI processors," Dig. 1982 Int. Test Conf., pp. 191-199, November, 1982; 46,47], or reasonableness of results [Mahmood, A., Lu, D. J. and McCluskey, E. J., "Concurrent fault detection using a watchdog processor and assertions," Proc. 1983 Int. Test Conf., pp. 622-628, October, 1983; Mahmood, A. Ersoz, a. and McCluskey, E.J., "concurrent system level error detection using a watchdog processor," Proc. 1985 Int. Test conf., pp. 145-152, November, 1985]. Using physical fault injection techniques, distributions of errors that could be detected using such types of information have been determined for some specific systems [Schmid, M., Trapp, R., Davidoff, A., and Masson, G., "Upset exposure by means of abstraction verification," Dig. of the 1982 Fault Tolerant Computing Symposium, pp. 237-244, June, 1982; Gunneflo, U., Karlsson, J., and Torin, J., "Evaluation of error detection schemes for using fault injection by heavy-ion radiation," Dig. of the 1989 Fault Tolerant Computing Symposium, pp. 340-347, June, 1989], and the performance of models of error monitoring techniques that could be realized in the form of watchdog processors have been analyzed [Blough, D., and Masson, G., "Performance analysis of a generalized concurrent error detection procedure," IEEE Trans. on Computers vol. 39, January, 1990.]. However, in contrast to the certification trail technique, a watchdog processor uses only a priori defined behavior checks, none of which is sufficient together with the input to the primary system to efficiently reproduce the output for direct comparison with that of the primary system.

Related to the watchdog processor approach is that of using executable assertions [Andrews, D., "Software fault tolerance through executable assertions," Rec. 12th Asilomar Conf. Circuits, Syst., Comput., pp. 641-645, 1978, November 6-8; Andrews, D., "Using

executable assertions for testing and fault tolerance," Dig. 9th Annu. Int. Sump. Fault-Tolerant Comput., pp. 102–105, 1979, June 20–22; Mahwood, A., Lu, D. J. and McCluskey E. J., "Concurrent fault detection using a watchdog processor and assertions," Proc. 1983 Int. Test Conf., pp. 622–628, October 1983]. An assertion can be defined as an invariant relationship among variables of a process. In a program, for examples, assertions can be written as logical statements and can be inserted into the code to signify that which has been predetermined to be invariably true at that point in the execution of the program. Assertions are based on a priori determined properties of the primary system or algorithm. This, however, again serves to distinguish executable assertion technique from the use of certification trails in that a certification trail is a key to the solution of a problem or the execution of an algorithm that can be utilized to efficiently and correctly produce the solution.

Algorithm-based fault tolerance [Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," IEEE Trans. on Computers, pp. 518–529, vol. C-33, June, 1984; Nair, V., and Abraham, J., "General linear codes for fault-tolerant matrix operations on processor arrays," Dig. of the 1988 Fault Tolerant Computing Symposium, pp. 180–185, June, 1988; "Fault tolerant FTT networks," Dig. of the 1985 Fault Tolerant Computing Symposium, June, 1985] uses error detecting and correcting codes for performing reliable computations with specific algorithms. This technique encodes data at a high level and algorithms are specifically designed or modified to operate on encoded data and produce encoded output data. Algorithm-based fault tolerance is distinguished from other fault tolerance techniques by three characteristics: the encoding of the data used by the algorithm; the modification of the algorithm to operate on the encoded data; and the distribution of the computation steps in the algorithm among computational units. It is assumed that at most one computational unit is faulty during a specified time period. The error detection capabilities of the algorithm-based fault tolerance approach are directly related to that of the error correction encoding utilized. The certification trail approach does not require that the data to be executed be modified nor that the fundamental operations of the algorithm be changed to account for these modifications. Instead, only a trail indicative of aspects of the algorithm's operations must be generated by the algorithm. As seen from the above examples, the production of this trail does not burden the algorithm with a significant overhead. Moreover, any combination of computational errors can be handled.

Recently Blum and Kannan [Blum, M., and Kannan, S., "Designing programs that check their work," Proceedings of the 1989 ACM Symposium on Theory of Computing, pp. 86–97, ACM Press, 1989] have defined what they call a program checker. A program checker is an algorithm which checks the output of an other algorithm for correctness and thus it is similar to an acceptance test in a recovery block. An example of a program checker is the algorithm developed by Tarjan [Tarjan, R. E., "Applications of path compression on balanced trees," J. ACM, pp. 690–715, October, 1979] which takes as input a graph and a supposed minimum spanning tree and indicates whether or not the tree actually is a minimum spanning tree. The Blum and Kannan checker is actually more general than this because it is allowed to be probabilistic in a carefully specified way. There are two main differences between this approach and the certification trail approach. First, a program checker may call the algorithm it is checking a polynomial number of times. In the certification trail approach the algorithm being checked is run once. Second, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input/output specification of a problem. The certification trail approach is explicitly algorithm being checked is run once. Second, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input-/output specification of a problem. The certification trail approach is explicitly algorithm oriented. In other words, a specific algorithm for a problem is modified to out put a certifications trail. This trail sometimes allows the second execution to be faster than any known program checkers for the problem. This is the case for the minimum spanning tree problem.

Other hardware and software fault tolerance and error monitoring techniques have been proposed and studied that might be thought of as bearing some resemblance to the certification trail approach. Extensive summaries and descriptions of these techniques can be found in the literature [Siewiorek, D., and Swarz, R., The theory and practice of reliable design, Digital Press, Bedford, Mass., 1982; Avizienis, A., "Fault tolerance by means of external monitoring of computer systems," Proceedings of the 1981 National Computer Conference, pp. 27–40, AFIPS Press, 1980; Johnson, B., Design and analysis of fault tolerant digital systems, Addison-Wesley, Reading, Mass., 1989; Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors—a survey," IEEE Trans. on Computers, vol. 37, pp. 160–174, February, 1988]. Examination of these techniques reveals, however, that in each case there are fundamental distinctions from the certification trail approach. In summary, the certification trail approach stands along in its employment of secondary algorithms/systems for the computation of an output for comparison that because of the availability of the trail not only proceeds in a more efficient manner than that of the primary but also can indicate whether the execution of the primary algorithm was correct.

Although the invention has been described in detail in the foregoing embodiments for the purpose of illustration, it is to be understood that such detail is solely for that purpose and that variations can be made therein by those skilled in the art without departing from the spirit and scope of the invention except as it may be described by the following claims.

What is claimed is:

1. A method for achieving fault tolerance in a computer system having at least a first central processing unit and a second central processing unit comprising the steps of:

executing a first algorithm in the first central processing unit on input so that a first output and a certification trail are produced;

executing a second algorithm in the second central processing unit on the input and on the certification trail so that a second output is produced, said second algorithm having a faster execution time than the first algorithm for a given input; and

comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

2. A method as described in claim 1 wherein the step of executing the second algorithm includes the step of determining whether the certification trail is in error.

3. A method as described in claim 2 including before the step of executing the first algorithm, there is the step of duplicating the input such that the input that is provided to the step of executing the first algorithm is also the input that is provided to the step of executing the second algorithm.

4. A method as described in claim 3 wherein the step of executing the first algorithm includes the step of determining whether the first output is in error.

5. A method as described in claim 4 wherein the step of executing the first algorithm includes the step of determining whether the second output is in error.

6. A method as described in claim 5 wherein the second algorithm generates the second output correctly when the second algorithm is executed by the second processing unit even if the certification trial produced by the first algorithm when the first algorithm is executed by the first processing unit is incorrect.

7. A method as described in claim 1 wherein the second algorithm is derived from the first algorithm.

8. A computer system comprising:

a first computer comprising:

a first memory,

a first central processing unit in communication with the memory,

a first input port in communication with the memory and the first central processing unit,

a first algorithm disposed in the first memory, said first algorithm produces a first output and produces a certification trail based on input received by the input port when the first algorithm is executed by the first central processor;

a second computer comprising a second memory,

a second central processing unit in communication with the second memory and the first central processing unit;

a second input port in communication with the second memory and the second central processing unit;

a second algorithm disposed in the second memory, said second algorithm produces a second output based on the input and the certification trail when the second algorithm is executed by the second central processing unit, said second algorithm having a faster execution time than the first algorithm for a given input; and

a mechanism for comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

9. A computer as described in claim 8 wherein the second algorithm generates the second output correctly when the second algorithm is executed by the second processing unit even if the certification trail produced by the first algorithm when the first algorithm is executed by the first processing unit is incorrect.

10. A computer system as described in claim 9 wherein the mechanism for comparing is a comparator.

11. An apparatus as described in claim 10 wherein the second algorithm is derived from the first algorithm.

12. A method for achieving fault tolerance in a central processing unit comprising the steps of:

executing a first algorithm in the central processing unit on input so that a first output and a certification trail are produced;

executing a second algorithm in the central processing unit on the input and on the certification trail so that a second output is produced, said second algorithm having a faster execution time than the first algorithm for a given input; and

comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

13. A method as described in claim 12 wherein the second algorithm generates the second output correctly when the second algorithm is executed by the processing unit even if the certification trail produced by the first algorithm when it is executed by the processing unit is incorrect.

14. A method as described in claim 13 wherein the second algorithm is derived from the first algorithm.

15. A computer comprising:

a memory,

a central processing unit in communication with the memory,

a first input port in communication with the memory and the central processing unit,

a first algorithm disposed in the memory, said first algorithm produces a first output and a certification trail based on input received by the input port when the input is executed by the central processing unit;

a second algorithm disposed in the memory, said second algorithm produces a second output based on the input and on at least a portion of the certification trail when the second algorithm is executed by the central processing unit, said second algorithm having a faster execution time than the first algorithm for a given input; and

a mechanism for comparing the first and second outputs such that an error result is produced if the first and second outputs are not the same.

16. A computer as described in claim 15 wherein the second algorithm generates the second output correctly when the second algorithm is executed by the processing unit even if the certification trail produced by the first algorithm when the first algorithm is executed by the processing unit is incorrect.

17. A computer as described in claim 16 wherein the mechanism for comparing is a comparator.

18. An apparatus as described in claim 15 wherein the second algorithm is derived from the first algorithm.

* * * * *

The Twenteth International Symposium on Fault-Tolerant Computing (1990)

Using Certification Trails to Achieve Software Fault Tolerance

Gregory F. Sullivan[1]
Gerald M. Masson[2]

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

We introduce a conceptually novel and powerful technique to achieve fault tolerance in hardware and software systems. When used for software fault tolerance, this new technique uses time and software redundancy and can be outlined as follows. In the initial phase, a program is run to solve a problem and store the result. In addition, this program leaves behind a trail of data which we call a *certification trail*. In the second phase, another program is run which solves the original problem again. This program, however, has access to the certification trail left by the first program. Because of the availability of the certification trail, the second phase can be performed by a less complex program and can execute more quickly. In the final phase, the two results are compared and if they agree the results are accepted as correct; otherwise an error is indicated. An essential aspect of this approach is that the second program must always generate either an error indication or a correct output even when the certification trail it receives from the first program is incorrect. We formalize the certification trail approach to fault tolerance and illustrate it by applying it to the fundamental problem of finding a minimum spanning tree. We discuss cases in which the second phase can be run concurrently with the first and act as a monitor. We compare the certification trail approach to other approaches to fault tolerance. Because of space limitations we have ommited examples of our technique applied to the Huffman tree, and convex hull problems. These can be found in the full version of this paper.

1 Introduction

In this paper we introduce a novel and powerful technique for achieving fault tolerance in systems. Although applicable to both hardware and software, we restrict our discussion of this technique in the following to software fault tolerance. To explain our new technique for software fault tolerance, we will first discuss a simpler fault tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so called time redundancy [14, 22]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming[8, 4] (in this case N=2), allows for the detection of errors caused by some faults in the software in addition to those caused by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

The technique we will describe is designed to achieve similar types of error detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. This data is chosen so that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains a error which causes an incorrect output and an incorrect trail of data to

[1] Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.
[2] Research parially supported by NASA Grant NSG 1442.

CH 2877-9/90/0000/0423/$01.00 – 1990 IEEE

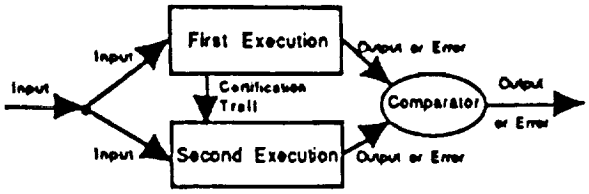


Figure 1: Certification trail method.

be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the data trail. Finally, it should be noted that in Figure 1 both executions can signal an error. These errors would include run-time errors such as divide-by-zero or non-terminating computation. In addition the second execution can signal error due to an incorrect certification trail.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 2.1 A problem P is formalized as a relation (that is, a set of ordered pairs). Let D be the domain (that is, the set of inputs) of the relation P and let S be the range (that is, the set of solutions) for the problem. We say an algorithm A solves a problem P iff for all $d \in D$ when d is input to A then an $s \in S$ is output such that $(d, s) \in P$.

Definition 2.2 Let $P : D \rightarrow S$ be a problem. Let T be the set of *certification trails*. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : D \rightarrow S \times T$ and $F_2 : D \times T \rightarrow S \cup \{\text{error}\}$. The functions must satisfy the following two properties:

(1) for all $d \in D$ there exists $s \in S$ and
there exists $t \in T$ such that
$F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in P$
(2) for all $d \in D$ and for all $t \in T$
either ($F_2(d, t) = s$ and $(d, s) \in P$) or
$F_2(d, t) = \text{error}$.

The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct. It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

The certification trail approach also allows for the detection of faults in software. As in 2-version programming, separate teams can write the algorithms for the first and second executions. Note that the specification now must include precise information describing the generation and use of the certification trail. Because of the additional data available to the second execution, the specifications of the two phases can be very different; similarly, the two algorithms used to implement the phases can be very different. This is illustrated by the convex hull example in the full paper. Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. This is illustrated by the minimum spanning tree example considered later. When significantly different algorithms are used, the probability that both algorithms will contain or be effected by faults which generate matching errors should be reduced. When very similar algorithms are used it is sometimes possible to save programming effort by sharing program code. While this reduces the ability to detect errors in the software it does not change the ability to detect transient hardware errors as discussed earlier.

Throughout this section we have assumed that our method is implemented with software; however, it is clearly possible to implement the certification trail technique by using dedicated hardware. It is also possible to generalize the basic two-level hierarchy of the certification trail approach as illustrated in Figure 1 to higher levels. Finally, we note that a wide variety of

approaches to software and hardware fault tolerance have been proposed which bear resemblances to the certification trail approach; we contrast our method to the most closely related ideas. A more comprehensive comparison appears in the full paper.

3 Minimum Spanning Tree Example

In this section we illustrate the use of the certification trail method by applying it to the minimum spanning tree problem. Because of space limitations we have ommited other applications, e.g., to the Huffman tree and the convex hull problems. It should be stressed here that we believe the technique has wide applicability and these problems were chosen simply for illustration.

The minimum spanning tree problem has been examined extensively in the literature and an historical survey is given in [11]. Our certification trail approach is applied to a variant of the Prim/Dijkstra algorithm [19, 9] as explicated in [24]. We will begin our discussion of the application of the certification trail approach to the minimum spanning tree problem with some preliminary definitions.

Definition 3.1 A *graph* $G = (V, E)$ consists of a *vertex* set V and an *edge* set E. An edge is an unordered pair of distinct vertices which we notate as, for example, $[v, w]$, and we say v is adjacent to w. A *path* in a graph from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $[v_i, v_{i+1}]$ is an edge for $i \in \{1, \ldots, k-1\}$. A path is a *cycle* if $k > 1$ and $v_1 = v_k$. An *acyclic* graph is a graph which contains no cycles. A *connected* graph is a graph such that for all pairs of vertices v,w there is a path from v to w. A *tree* is an acyclic and connected graph.

Definition 3.2 Let $G = (V, E)$ be a graph and let w be a positive rational valued function defined on E. A *subtree* of G is a tree, $T(V', E')$, with $V' \subseteq V$ and $E' \subseteq E$. We say T *spans* V' and V' is spanned by T. If $V' = V$ then we say T is a spanning tree of G. The weight of this tree is $\sum_{e \in E'} w(e)$. A minimum spanning tree is a spanning tree of minimum weight.

3.0.1 Data structures and supported operations

Before we discuss the minimum spanning tree algorithm, we must describe the properties of the principle data structure that are required. Since many different data structures can be used to implement the algorithm, we initially describe abstractly the data that can be stored by the data structure and the operations that can be used to manipulate this data. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set; however, at all times, the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, k is a key value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i, k) < (i', k')$ iff $k < k'$ or $(k = k'$ and $i < i')$. Our data structure should support a subset of the following operations.

member(i, h) returns a boolean value of true if h contains an ordered pair with item number i, otherwise returns false.

insert(i, k, h) adds the ordered pair (i, k) to the set h.

delete(i, h) deletes the unique ordered pair with item number i from h.

changekey(i, k, h) is executed only when there is an ordered pair with item number i in h. This pair is replaced by (i, k).

deletemin(h) returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If h is the empty set then the token "empty" is returned.

predecessor(i, h) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned.

Many different types and combinations of data structures can be used to support these operations efficiently. In our case, we will actually use two different data structure methods to support these operations.

One method will be used in the first execution of the algorithm and another, faster and simpler, method will be used in the second execution. The second method relies on a trail of data which is output by the first execution.

3.0.2 MINSPAN algorithm

Before discussing precise implementation details for these methods we present the overall algorithm used in both executions. Pidgin code for this algorithm appears below. In addition, Figure 2 illustrates the execution of the algorithm on a sample graph and the table below records the data structure operations the algorithm must perform when run on the sample graph. The first column of the table gives the operations except *member* and with the parameter h dropped to reduce clutter. The second column gives the evolving contents of h. The third column records the ordered pair deleted by the *deletemin* operation. The fourth column records the certification trail corresponding to these operations and is further discussed below.

The algorithm uses a "greedy" method to "grow" a minimum spanning tree. The algorithm starts by choosing an arbitrary vertex from which to grow the tree. During each iteration of the algorithm a new edge is added to the tree being constructed. Thus, the set of vertices spanned by the tree increases by exactly one vertex for each iteration. The edge which is added to the tree is the one with the smallest weight. Figure 2 shows this process in action. Figure 2(a) shows the input graph, Figures 2(b) through 2(e) show several stages of the tree growth and Figure 2(f) shows the final output of the minimum spanning tree. The solid edges in Figures 2(b) through 2(e) represent the current tree and the dotted edges represent candidates for addition to the tree.

To efficiently find the edge to add to the current tree the algorithm uses the data structure operations described above. As soon as a vertex , say v, is adjacent to some vertex which is currently spanned it is inserted in the set h. The key value for v is the weight of the minimum weight edge between v and some vertex spanned by the current tree. The array element *prefer*(v) is used to keep track of this minimum weight edge. As the tree grows, information is updated by operations such as *insert*(i, k, h) and *changekey*(i, k, h). The *deletemin*(h) operation is used to select the next vertex to add to the span of the current tree. Note, the algorithm does not explicitly keep a set of edges

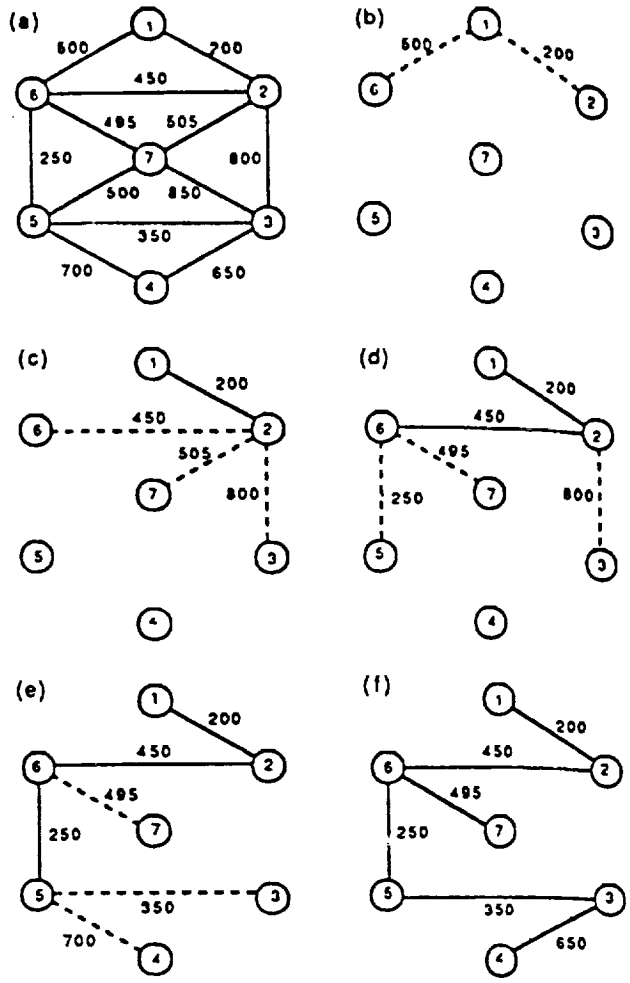


Figure 2: Example for minimum spanning tree algorithm.

representing the current tree. Implicitly, however, if (v, k) is returned by *deletemin* then *prefer*(v) is added to the current tree.

3.0.3 First execution of MINSPAN

In the first execution of the algorithm, the MINSPAN code is used and the principle data structure is implemented with a balanced search tree such as an AVL tree [1], a red-black tree [12], or a b-tree [5]. In addition, an array of pointers indexed from 1 to n is used. The balanced search tree stores the ordered pairs in h and is based on the total order described earlier. The array of pointers is initially all nil. For each item i, the ith pointer of the array is used to point to the lo-

Algorithm MINSPAN(G, *weight*)
Input: Connected graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ with edge weights.
Output: Spanning tree of G which has minimum weight

```
1  CHOOSE root ∈ V
2  FOR ALL u ∈ V, key(u) := ∞ END FOR
3  h := ∅; v := root
4  WHILE v ≠ empty DO
5   key(v) := −∞
6   FOR EACH [v, w] ∈ E DO
7    IF weight([v, w]) < key(w) THEN
8     key(w) := weight([v, w]); prefer(w) := [v, w]
9     IF member(w, h) THEN changekey(w, key(w), h)
10     ELSE insert(w, key(w), h) END IF
11   END IF
12  END FOR
13  (v, k) := deletemin(h)
14 END WHILE
15 FOR ALL u ∈ V − {root}, OUTPUT(prefer(u))
END MINSPAN
```

Figure 3: Code for MINSPAN Algorithm

Operation	Set of Ordered Pairs	Trail
insert(2,200)	(2,200)	smallest
insert(6,500)	(2,200),(6,500)	2
deletemin	(6,500)	
insert(3,800)	(6,500),(3,800)	6
changekey(6,450)	(6,450),(3,800)	smallest
insert(7,505)	(6,450),(7,505),(3,800)	6
deletemin	(7,505),(3,800)	
insert(5,250)	(5,250),(7,505),(3,800)	smallest
changekey(7,495)	(5,250),(7,495),(3,800)	5
deletemin	(7,495),(3,800)	
changekey(3,350)	(3,350),(7,495)	smallest
insert(4,700)	(3,350),(7,495),(4,700)	7
deletemin	(7,495),(4,700)	
changekey(4,650)	(7,495),(4,650)	7
deletemin	(4,650)	
deletemin		
deletemin		

Table 1: Data structure operations and certification trail for MINSPAN

cation of the ordered pair with item number i in the balanced search tree. If there is no such ordered pair in the tree then the ith pointer is nil. This array allows rapid execution of operations such as *member*(i, h) and *delete*(i, h).

The certification trail is generated during the first execution as follows: When CHOOSE *root* $\in V$ is executed in the first step, the vertex which is chosen is output. Also, each time *insert*(i, k, h) or *changekey*(i, k, h) are executed, *predecessor*(i, h) is executed afterwards, and the answer returned is output. This is illustrated in column labeled "Trail" in the table above.

3.0.4 Second execution of MINSPAN

The second execution of the algorithm also uses the MINSPAN code; however, the CHOOSE construct and the data structure operations are implemented differently than in the first execution. The CHOOSE is performed by simply reading the first element of the certification trail. This guarantees the same choice of a starting vertex is made in both executions. Figure 4 depicts the principle data structure used which we call an *indexed linked list*. The array is indexed from 1 to n and contains pointers to a singly linked list which represents the current contents of h. Each element in the list stores an ordered pair in h except the head of the list which contains the special ordered pair $(0, -\text{INF})$. The list is organized such that a traversal from the head gives the sorted ordering of the current contents of h from smallest to largest. The ith element of the array points to the node containing the ordered pair with the item number i if it is present in h; otherwise, the pointer is nil. The 0th element of the array points to the node containing $(0, -\text{INF})$. Initially, the array contains nil pointers except the 0th element. We now show how to implement the data structure operations.

To perform *insert*(i, k, h), it is necessary to read the next value in the certification trail. This value, say j, is the item number of the ordered pair which is the predecessor of (i, k) in the current contents of h. A new linked list node is allocated and the trail information is used to insert the node into the data structure. Specifically, the jth array pointer is traversed to a node in the linked list, say Y. (If j = "smallest" then the 0th array pointer is traversed.) The new node is inserted in the list just after node Y and before the next node in the linked list (if there is one). The data field in the new node is set to (i, k) and the ith pointer of the array is set to point to the new node. Figure

4 shows the insertion of $(7, 505)$ into the data structure given that the certification trail value is 6. Figure 3(a) is before the insertion and Figure 3(b) is after the insertion.

When the *insert* operation is performed, some checks must be conducted. First, the ith array pointer must be nil before the operation is performed. Second, the sorted order of the pairs stored in the linked list must be preserved after the operation. That is, if (i', k') is stored in the node before (i, k) in the linked list and (i'', k'') is stored after (i, k), then $(i', k') < (i, k) < (i'', k'')$ must hold in the total order. If either of these checks fails then execution halts and "error" is output.

To perform *delete*(i, h) the ith array pointer is traversed and the node found is deleted from the linked list. Next, the ith array pointer is set to nil. Figure 4 shows the deletion of item number 7 if one considers Figure 3(a) as depicting the data structure before the operation and Figure 3(b) depicting it afterwards. When the *delete* operation is performed one check is made. If the ith array pointer is nil before the operation then the execution halts and "error" is output.

To perform *changekey*(i, k, h) it suffices to perform *delete*(i, h) followed by *insert*(i, k, h). Note, this means the next item in the certification trail is read. Also, the checks associated with both these two operations are performed and the execution halts with "error" output if any check fails.

To perform *deletemin*(h) the 0th array pointer is traversed. to the head of the list and the next node in the list is accessed. If there is no such node then "empty" is returned and the operation is complete. Otherwise, suppose the node is Y and suppose it contains the ordered pair (i, k), then the node Y is deleted from the list, the ith array pointer is set to nil, and (i, k) is returned.

Lastly, to perform *member*(i, h) the ith array pointer is examined. If it is nil then false is returned, otherwise, true is returned. The *predecessor*(i, h) operation is not used in the second execution.

This completes the description of the second execution. To show that what we have described is a correct implementation of the certification trail method requires a proof. The proof has several parts of varying difficulty. First, one must show that if the first execution is fault-free then it outputs a minimum spanning tree. Second, one must show that if the first and second executions are fault-free then they both output the same minimum spanning tree. Both these parts of the proof are not difficult to show.

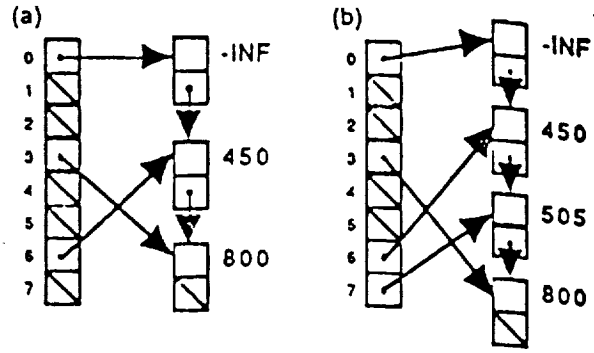


Figure 4: Example of the data structure used in the second execution of MINSPAN.

The third more subtle part of the proof deals with the situation in which only the second execution is fault-free. This means an incorrect certification trail may be generated in the first execution. In this case, we must show that the second execution outputs either the correct minimum spanning tree or "error". The checks that were described above have been carefully designed to assure precisely this property by detecting any errors that would prevent the execution from generating the correct output. Because of space restrictions we will not give the proof here.

3.0.5 Time complexity comparisons of the two executions

In the first execution each data structure operation can be performed in $O(\log(n))$ time where $|V| = n$. There are at most $O(m)$ such operations and $O(m)$ additional time overhead where $|E| = m$. Thus, the first execution can be performed in $O(m \log(n))$ We note that this algorithm does not achieve the fastest known asymptotic time complexity which appears in [10]. However, the algorithm we have presented has a significantl smaller constant of proportionality which makes it competitive for reasonably sized graphs. In addition, it provides us with a relatively simple and illustrative example of the use of a certification trail. It should be mentioned that we have developed a more complex certification trail solution for an asymptotically faster minimum spanning tree algorithm which uses fibonacci heaps.

In the second execution each data structure operation can be performed in $O(1)$. There are still at most $O(m)$ such operations and $O(m)$ additional time overhead. Hence, the second execution can be performed in $O(m)$ time. In other words, because of the availability of the certification trail, *the second execution is performed in linear time.* There are no known $O(m)$ time algorithms for the minimum spanning tree problem. Komlòs was able to show that $O(m)$ comparisons suffice to find the minimum spanning tree. However, there is no known $O(m)$ time algorithm to actually find and perform these comparisons. Even the related "verification" problem has no known linear time solution. In the verification problem the input consists of an edge weighted graph and a subtree. The ouput is "yes" if the subtree is the minimum spanning tree and "no" otherwise. The best known algorithm for this problem was created by Tarjan [25] and has the nonlinear time complexity of $O(m\alpha(m,n))$, where $\alpha(m,n)$ is a functional inverse of Ackerman's function. The fact that the data in a certification trail enables a minimum spanning tree to be found in linear time is, we believe, intriguing, significant, and indicative of the great promise of the certification trail technique.

3.1 Concurrency of Executions

In some cases, it is possible to start the second execution before the first execution has terminated. This is a highly desirable capability when additional hardware is available to run the second execution (for example, with multiprocessor machines, or machines with coprocessors or hardware monitors).

In the case of the minimum spanning tree problem, the two executions can be run concurrently. It is only necessary for the second execution to read the certification trail as it is generated - one item number at a time. Thus there is a slight time lag in the second execution. This potential for concurrecy has been found in other problems we have examined, e.g., the Huffman tree problem.

An additional opportunity for overlapping execution occurs when the system has a dedicated comparator. In this case it is sometimes possible for the two executions to send there output to the comparator as they generate it. For example, this can be done in the minimum spanning tree problem where the edges of the tree can be sent individually as they are discovered by both executions.

4 Comparison of Techniques

The certification trail approach, whether implemented in hardware or software or some combination thereof, has resemblances with other fault tolerant techniques that have been previously proposed and examined, but in each case there are significant and fundamental distinctions. These distinctions are primarily related to the generation and character of the certification trail and the manner in which the secondary algorithm or system uses the certification trail to indicate whether the execution of the primary system or algorithm was in error and/or to produce an output to be compared with that of the primary system.

To begin, we compare the certification trail approach to N-version programming[8, 4]. This approach specifies that N different implementations of an algorithm be independently executed with subsequent comparison of the resulting N outputs. There is no relationship among the executions of the different versions of the algorithms other than they all use the same input; each algorithm is executed independently without any information about the execution of the other algorithms. In marked contrast, the certification trail approach allows the primary system to generate a trail of information while executing its algorithm that is critical to the secondary system's execution of its algorithm. In effect, N-version programming can be thought of relative to the certification trail approach as the employment of a *null trail.*

A software/hardware fault tolerance technique called the recovery block approach [20, 2, 17] uses acceptance tests and alternative procedures to produce what is to be regarded as a correct output from a program. When using recovery blocks, a program is viewed as a being structured into blocks of operations which after execution yield outputs which can be tested in some informal sense for correctness. The rigor, completeness, and nature of the acceptance test is left to the program designer [2]. Indeed, formal methodologies for the definition and generation of acceptance tests have thus far not been fully established. Regardless, the certification trail notion of a secondary system that receives the same input as the primary system and executes an algorithm that takes advantage of this trail to efficiently produce the correct output and/or to indicate that the execution of the first algorithm was correct does not fall into the category of an acceptance test.

Recently Blum and Kannan[7] have defined what they call a *program checker.* A program checker is

an algorithm which checks the output of another algorithm for correctness and thus it is similar to an acceptance test in a recovery block. An example of a program checker is the algorithm developed by Tarjan[25] which takes as input a graph and a supposed minimum spanning tree and indicates whether or not the tree actually is a minimum spanning tree. The Blum and Kannan checker is actually more general than this because it is allowed to be probabilistic in a carefully specified way. There are two main differences between this approach and the certification trail approach. First, a program checker may call the algorithm it is checking a polynomial number of times. In our approach the algorithm being checked is run once. Second, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input/output specification of a problem. The certification trail approach is explicitly algorithm oriented. In other words, a specific algorithm for a problem is modified to output a certification trail. This trail sometimes allows the second execution to be faster than any known program checkers for the problem. This is the case for the minimum spanning tree problem.

Space limitations preclude comparisons with the following other relevant techniques: watchdog processors [18, 6], algorithm based fault tolerance [13], executable assertions [3].

5 Concluding Discussion

We have presented a new, powerful fault tolerant computing technique called the certification trail approach. Our description of this technique has been only in terms of applications to software fault tolerance, but the certification trail approach can also be implemented with hardware. We have illustrated the certification trail technique by applying it to a minimum spanning tree algorithm. The full version of this paper includes applications to a Huffman tree algorithm, and a convex hull algorithm. It should be understood that the approach is in no way limited to these algorithms. We believe that our consideration of these algorithms gives insight into the significance and desirability of the approach. We have found several other algorithms to which our techniques apply including an algorithm for the shortest path problem and we believe the technique will be widely applicable. We have also examined the general problem of "certifying" data structure operations as discussed above and have proven results for additional data structures. These results are important because they allow the certification trail approach to be applied to any algorithm which uses one of these data structures.

In the problem discussed an asymptotic speed up was achieved between the first execution and the second execution which was greater than any constant factor. We note, however, even if the speed up were only by a constant factor, it would still make sense to use the technique because execution time would be saved. We also note that the certification trail technique can be used in conjunction with other software fault tolerance techniques. For example, multiple algorithms can be developed which generate and read multiple (but different) certification trails. Further, these algorithms could be written by separate teams of individuals. A general architecture for the interaction of these algorithms is an important research topic. For example, a "cascade" of algorithms numbered from 1 to N could be designed such that algorithm i sends a certification trail to $i+1$ which allows $i+1$ to run faster than i. When errors are detected, other versions of algorithms can be invoked which may use an earlier certification trail or ignore it. The ideas developed in recovery blocks and N-version programming among others could be used as guidance in exploring such issues.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Andrews, D., "Using executable assertions for testing and fault tolerance," *Dig. 9th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 102-105, 1979, June 20-22.

[4] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[5] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[6] Blough, D., and Masson, G., "Performance analysis of a generalized concurrent error detection procedure," *IEEE Trans. on Computers* vol. 39, Jan., 1990.

[7] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[8] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[9] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, Sept., 1959.

[10] Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E., "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica 6*, pp. 109-122, 2, 1986.

[11] Graham, R. L., and Hell, P., "On the history of the minimum spanning tree problem," *Ann. Hist. Comput.*, pp. 43-47, Jan., 1985.

[12] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[13] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

[14] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[15] Kane, J.R. and Yau, S.S., "Concurrent software fault detection," *IEEE Trans. Software Eng.* , vol. SE-1, pp. 87-99, March 1975.

[16] Komlòs, J., "Linear verification for spanning trees", *Proceedings of the 1984 Symposium on Foundations of Computing*, pp. 201-206, IEEE Computer Society Press, 1984.

[17] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[18] Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors - a survey," *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb., 1988.

[19] Prim, R. C., "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, pp. 1389-1401, Nov., 1957.

[20] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[21] Shen, J.P. and Schuette, M.A., "On-line self-monitoring using signatured instruction streams," *Proc. 1983 Int. Test Conf.*,, pp. 275-282, Oct., 1983.

[22] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[23] Sridhar, T. and Thatte, S.M., "Concurrent checking of program flow in VLSI processors," *Dig. 1982 Int. Test Conf.*, pp. 191-199, Nov., 1982.

[24] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[25] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[26] Tomas, S. P. and Shen, J. P., "A roving monitoring processor for detection of control flow errors in multiple processor systems," *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput.*, pp.531-539, Oct., 1985.

[27] Yau, S.S, and Chen, F.-C., "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126-137, March 1980.

ORIGINAL PAGE IS OF POOR QUALITY

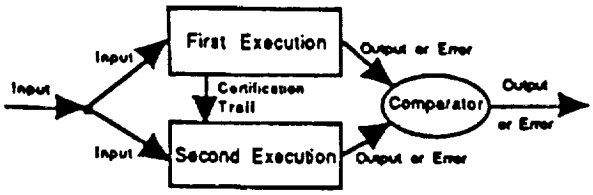


Figure 1: Certification trail method.

output such that $(d, s) \in P$.

Definition 2.2 Let $P : D \rightarrow S$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : D \rightarrow S \times T$ and $F_2 : D \times T \rightarrow S \cup \{error\}$. T is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in D$ there exists $s \in S$ and
there exists $t \in T$ such that
$F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in P$
(2) for all $d \in D$ and for all $t \in T$
either $(F_2(d, t) = s$ and $(d, s) \in P)$ or
$F_2(d, t) =$ error.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. The definitions above assure that the error-detection capability of the certification-trail approach is similar to that obtained with the simple time-redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

Observant readers of our earlier paper [11] in which we introduced the notion of a certification trail might have noticed that our certification-trail solution for the min-spanning tree was generalizable. The generalized technique allows one to generate a certification trail for many algorithms which use a balanced binary tree data structure. However, the technique relies on the efficient execution of the predecessor operation and some data structures such as heaps cannot execute the predecessor operation efficiently. The techniques described in this paper are even more general and powerful, and they do apply to heaps.

The degree of diversity or independence achieved when using certification trails depends on how they are used. A fuller discussion of this and of the relationship between certification trails and other approaches to software fault tolerance is contained in the expanded version of [11]. This current paper presents asymptotic analysis which shows that the certification-trail approach is desirable even when the overhead of generating the certification-trail is included. We are currently working on an experimental analysis of the method and initial results are quite promising.

3 Answer-Validation Problem for Abstract Data Types

Our general approach to applying certification trails uses the concept of an abstract data type. Some examples of abstract data types are given later in this paper. Here we mention some important common properties and give a short illustration. Each abstract data type has a well defined data object or set of data objects, and each abstract data type has a carefully defined finite collection of operations that can be performed on its data object(s). Each operation takes a finite number of arguments (possibly zero), and some but not all operations return answers. An example of an abstract data type is a priority queue. The data object for a priority queue is an ordered pair of the form (i,k) where i is an item number and k is a key value. A priority queue has two operations: insert(i,k) and delmin. The insert operation has two arguments: item number i and key value k. The insert operation does not return an answer. The delmin operation has no arguments, but it does return an answer. The precise semantics of these operations are given later in this paper.

For each abstract data type we define an *answer-validation* problem. Intuitively, the answer validation problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on the abstract data type. More formally, the input to the answer-validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. In addition, the sequence contains the supposed answers for each of the operations which return answers. In particular, each supposed answer is paired with the operation that is supposed to return it. Examples of such inputs are given in the columns labelled "Operation" and "Answer" of table 1 and table 2.

The output for the answer-validation problem is the word "correct" if the answers given in the input match the answers that would be generated by actually performing the operations. The output is the word "incorrect" if the answers do not match. It is also useful to allow the output word to say "ill-formed". This output is used if the sequence of operations is ill-formed, e.g., an operation has too many arguments or an argument refers to an inappropriate object.

Certification Trails for Data Structures

Gregory F. Sullivan[1]
Gerald M. Masson[2]

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault detection and fault tolerance [11]. In this paper, we significantly generalize the applicability of the certification trail technique. Previously, certification trails had to be customized to each algorithm application, but here we develop trails appropriate to wide classes of algorithms. These certification trails are based on common data-structure operations such as those carried out using balanced binary trees and heaps. Any algorithm using these sets of operations can therefore employ the certification trail method to achieve software fault tolerance. To exemplify the scope of the generalization of the certification trail technique provided in this paper, constructions of trails for abstract data types such as priority queues and union-find structures will be given. These trails are applicable to any data-structure implementation of the abstract data type. It will also be shown that these ideas lead naturally to monitors for data-structure operations.

Keywords: Software fault tolerance, error monitoring, certification trails, design diversity, data structures.

1 Introduction

In this paper we significantly generalize the novel and powerful certification-trail technique for achieving fault tolerance in systems that was introduced in [11]. Although applicable to both hardware and software, we restrict our discussion of the certification-trail technique in the following to software fault tolerance. To explain the essence of the certification-trail technique for software fault tolerance, we will first discuss a simpler fault-tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so-called time redundancy [8, 10]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct. The second possibility, of undetected faults, occurs when the output of the execution is unaffected by the faults.

The certification-trail technique is designed to obtain similar types of error-detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. This data is chosen so that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected in the data trail.

2 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 2.1 A problem P is formalized as a relation, i.e., a set of ordered pairs. Let D be the domain (that is, the set of inputs) of the relation P and let S be the range (that is, the set of solutions) for the problem. We say an algorithm A solves a problem P iff for all $d \in D$ when d is input to A then an $s \in S$ is

[1]Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.
[2]Research partially supported by NASA Grant NSG 1442.

434 — 439

The answer-validation problem is similar to the idea of an acceptance test which is used in the recovery-block approach [9, 2] to software fault tolerance. The main difference is that an answer-validation problem is dependent upon a sequence of answers, not just an individual answer. Hence, if an incorrect answer appears in the sequence, it may not be detected immediately. It is guaranteed, however, that an incorrect answer will be detected at some point during the processing of the entire sequence. By allowing for this latency in detection, it is possible to create a much more efficient procedure for solving the answer-validation problem.

In this paper we shall solve the validation problem for two abstract data types. In the full version of this paper we solve the answer-validation problem for more general data types [12].

The most important aspect of the answer-validation problem is that it is often possible to check the correctness of the answers to a sequence of operations much more quickly than actually calculating what the answers should be from scratch. In other words, the answer-validation problem has a smaller time complexity than the original abstract-data-type problem. For example, to calculate the answers to a sequence of n priority-queue operations takes $\Omega(n \log(n))$ time, however it is possible to check the correctness of the answers in only $O(n)$ time. This speedup is very useful in fault-detection applications.

It is possible to run an answer-validation algorithm for some abstract data type concurrently with some algorithm which uses the abstract data type. The answer-validation algorithm could act as a monitor making sure that all interactions with the abstract data type are handled correctly. This is valuable because many algorithms spend a large fraction of their time operating on abstract data types. Note, the overhead of this monitor is less than the overhead of actually performing the data-type operations a second time.

One possible application of the answer-validation problem occurs when it is used in conjunction with a repairable data structure which allows for repair but does not automatically attempt to detect faults [16]. Suppose an abstract data type is implemented with a repairable data structure. One can use an answer-validation procedure to detect errors in the answers generated by the abstract data type. When an error is detected, a repair of the data structure can be attempted. In some cases, recovery and continued execution will be possible.

In the next section, we will show how to create certification trails for programs which use abstract data types when those data types have efficient solutions for their answer-validation problems.

4 Schema for using Certification Trails

Suppose that we have developed an efficient solution to the answer-validation problem for some abstract data type. By efficient we mean the time complexity of the answer-validation problem is smaller than the time complexity of the original abstract-data-type problem. Further, suppose that we wish to run an algorithm, say A, which uses that abstract data type. To apply the certification trail method we can use the following schema to yield the two executions:

First Execution:

Execute algorithm A.
Each time an abstract-data-type operation is performed, append to the certification trail the identity of the operation, the arguments and the answer.

Second execution:

Phase One:
Validate the correctness of the operations and supposed answers given in the certification trail. If the validation returns "incorrect" or "ill-formed" then output "error" and stop. Otherwise, continue.

Phase Two:
Execute algorithm A.
Each time an abstract-data-type operation is performed, read the next entry in the certification trail. Make sure that the operation and the arguments in the certification trail agree with those requested in the algorithm. If not output "error" and stop. Otherwise, use the answer given in the certification trail and continue.

In the final step the outputs from the two executions are compared and the output is accepted or an error is signaled. This schema can yield execution times which are significantly faster than the execution time obtained by running algorithm A twice, yet these two methods give similar fault detection capabilities. That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct. Note, the first execution can be slower than a simple execution of algorithm A since it must output a certification trail. However, the second execution can be significantly faster than a simple execution of the algorithm since the interactions with the abstract data type take less time overall. The net effect can be a major speedup.

Suppose an algorithm uses multiple abstract data types and suppose there are efficient answer-validation algorithms for each of these abstract data types. It is easy to see how our method generalizes. We can leave behind a generalized certification trail which consists of a separate certification trail for each of the abstract data types. The effect on the speedup of the second execution will be cumulative.

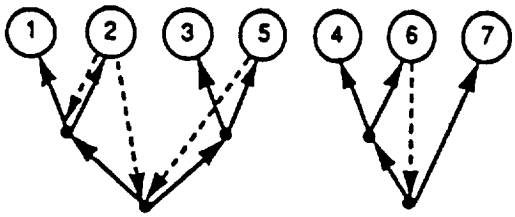


Figure 2: Union Tree and with Find Edges

5 Answer Validation for Disjoint-Set Union

As our first example we will discuss the disjoint-set union problem. This problem concerns a dynamic collection of sets in which pairs of sets can be combined to yield new sets. The underlying universe of set elements consists of the integers from 1 to n inclusive. Also, the universe of set names consists of the integers from 1 to n inclusive. There are three operations that can be performed:

create(A,x) creates a singleton set named A which contains element x. Since sets must be disjoint we require that x not already be in some set.

union(A,B) creates a new set which is the union of the sets named A and B. This new set is called A and the set named B becomes undefined. It is required that the sets named A and B are originally defined and are disjoint.

find(x) returns the name of the set which contains element x. It is required that x be a member of some unique set.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

In table 1 we give an example of a sequence of disjoint-set-union operations together with the answers for find operations. In addition, the collection of sets is depicted as it is changed by the operations. For simplicity, in this example each set name corresponds to the integer originally contained in the set when it is created. Sets are listed by first giving the name of the set followed by a colon and then the contents of the set.

The disjoint-set-union problem is a classic problem which has many applications [4] such as the off-line

Operation	Answer	Status of sets
create(1,1)		1:{1}
create(2,2)		1:{1},2:{2}
union(1,2)		1:{1,2}
find(2)	1	
create(3,3)		1:{1,2},3:{3}
create(4,4)		1:{1,2},3:{3},4:{4}
create(5,5)		1:{1,2},3:{3},4:{4},5:{5}
union(5,3)		1:{1,2},4:{4},5:{3,5}
union(5,1)		4:{4},5:{1,2,3,5}
find(2)	5	
find(5)	5	
create(6,6)		4:{4},5:{1,2,3,5},6:{6}
union(4,6)		4:{4,6},5:{1,2,3,5}
create(7,7)		4:{4,6},5:{1,2,3,5},7:{7}
union(4,7)		4:{4,6,7},5:{1,2,3,5}
find(6)	4	

Table 1: Sequence of operations for a Disjoint Set Union

min problem, connected components, least-common ancestors, and equivalence of finite automata. Of particular interest is the time-complexity of performing a sequence of operations. Let us say the total number of operations is m, which is assumed to be greater than or equal to n. Recall, n is the number of set elements and set names.

Tarjan gave the tight upper bound of $O(m\alpha(m,n))$ [13, 14] for this problem. The α refers to the inverse of Ackermann's function which is a very slowly growing function. His solution and earlier solutions used a path-compression heuristic [15]. Fredman and Saks gave a lower bound of $\Omega(m\alpha(m,n))$ [5] in a general cell-probe model. Gabow and Tarjan show how to solve some important special cases of this problem in $O(m)$ time [6].

We now consider the answer-validation problem for the disjoint-set-union data type. We will show that this problem can be solved in $O(m)$ time where m is the number of operations. Note, this time complexity is superior to the complexity of actually performing the sequence of operations as discussed above. One method for solving this problem in $O(m)$ time uses the powerful techniques of Gabow and Tarjan [6]. However, we shall present a simpler method with a small constant of proportionality that is tailored to this problem.

To solve this problem we will build a forest based on the union operations in the sequence. In addition, we shall add edges to this forest based on the find operations. As a final step we will perform a traversal of the forest and perform appropriate checks. The solid edges in figure 2 indicate the forest we would build for

the set of operations given in table 1. The dashed edges indicate the edges we would add to the forest based on the find operations.

Algorithm for Answer Validation for Disjoint-Set Union

Input: sequence of m operations together with arguments and supposed answers for the disjoint-set union data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Type *treenode* has fields left and right. Type *treeleaf* contains a list of pointers such that each pointer points to a treenode or a treeleaf. Array *activeset* is indexed by set name. Each array element is a pointer to a treenode or a treeleaf. Array *whereis* is indexed by an element number. Each array element is a pointer to a treeleaf. Initially, all pointers are nil and lists are null.

In the first phase of the algorithm we process each operation as it appears serially using the following rules:

create(A,x): If activeset[A] or whereis[x] are non-nil then output "ill-formed" and stop. Otherwise, allocate a treeleaf and set activeset[A] and whereis[x] to the allocated node.

union(A,B): If activeset[A] or activeset[B] are nil then output "ill-formed" and stop. Otherwise, allocate a treenode and set left to activeset[A] and right to activeset[B]. Next set activeset[A] to the treenode and set activeset[B] to nil.

find(x) A: (where A is the supposed answer to the find.) If whereis[x] is nil then output "ill-formed". Otherwise, whereis[x] points to some treeleaf. Call it tleaf. If activeset[A] is nil then output "ill-formed". Otherwise, activeset[A] points to some treeleaf or treenode. Call it t. Add a pointer to t to the list of pointers contained in treeleaf.

In the second phase of the algorithm we shall traverse the structure we have built.

Scan thru the array activeset to find non-nil pointers. It is not hard to see that each non-nil pointer points to the root of a tree made up of nodes of type tnode and tleaf. The tree uses the edges in the left and right fields of tnode.

For each such tree perform a depth-first search. Whenever the search reaches a node of type tleaf traverse the list of pointers that it contains. Check that each pointer points to a node which is currently on the stack which is used to perform the depth-first search. This is equivalent to checking that each pointer in tleaf points to a node which is an ancestor of tleaf in the tree.

If some pointer does not point to an ancestor then output "incorrect" and stop. Otherwise, output "correct" and stop.

Theorem 5.1 *The algorithm for answer validation of the disjoint-set-union abstract data type is correct.*

Theorem 5.2 *The answer validation algorithm for disjoint set union has a time complexity of $O(m)$ for processing a sequence of m operations.*

We omit these two theorems which overall are not difficult to show. We comment on one aspect of implementation. In the second phase of the answer validation algorithm it is necessary to determine if certain nodes are on the stack during the tree traversal. This can be done efficiently as follows: First, each treenode and each treeleaf can be assigned a unique identifier in the range 1 to m as it is allocated. Next, a boolean vector of size m indexed by the unique identifiers described above can be allocated. This vector can be used to keep track of which nodes are on the stack during tree traversal by turning bits on and off. This modified tree traversal algorithm still takes $O(m)$ time.

6 Generalized Priority Queue

We now describe a somewhat general abstract data type. We will solve the answer validation problem for restricted versions of this data type. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, k is a key value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i,k) < (i',k')$ iff $k < k'$ or $(k = k'$ and $i < i')$. The abstract data types we will consider support a subset of the following operations.

member(i) returns a boolean value of true if the set contains an ordered pair with item number i, otherwise returns false.

insert(i,k) adds the ordered pair (i,k) to the set. We require that no other pair with item number i be in the set.

delete(i) deletes the unique ordered pair with item number i from the set. We require that a pair with item number i be in the set initially.

changekey(i,k) is executed only when there is an ordered pair with item number i in the set. This pair is replaced by (i,k).

T	Operation	Answer	Validation stack
1	insert(6,300)		
2	insert(2,404)		
3	insert(3,250)		
4	deletemin	(3,250)	(3,250,4)
5	insert(10,248)		
6	insert(12,245)		
7	insert(4,260)		
8	deletemin	(12,245)	(12,245,8),(3,250,4)
9	insert(13,140)		
10	insert(5,142)		
11	deletemin	(13,140)	(13,140,11),(12,245,8),(3,250,4)
12	deletemin	(5,142)	(5,142,12),(12,245,8),(3,250,4)
13	deletemin	(10,248)	(10,248,13),(3,250,4)
14	deletemin	(4,260)	(4,260,14)

Table 2: Sequence of Priority Queue operations illustrating answer validation algorithm

deletemin (or deletemax) returns the ordered pair which is smallest (or largest) according to the total order defined above and deletes this pair. If the set is empty then the token "empty" is returned.

min (or max) returns the ordered pair which is smallest (or largest) according to the total order defined above. If the set is empty then the token "empty" is returned.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

Many different types and combinations of data structures can be used to support different subsets of these operations efficiently.

7 Answer Validation for Priority Queue

We will first consider the priority-queue abstract data type which allows only two operations: insert and deletemin. An example of a sequence of such operations appears in table 2. Many different data structures can be used to implement priority queues including heaps [17], balanced search trees such as AVL trees [1], red-black trees [7], or b-trees [3]. It is possible to process a sequence of $O(n)$ operations in $O(n \log(n))$ time using the data structures above. Furthermore, there is a lower bound of $\Omega(n \log(n))$ because it is possible to sort using a priority queue. Remarkably, the answer-validation problem can be solved using only $O(n)$ time, as documented below.

Each operation is time-stamped, i.e., the operations are assigned integers sequentially starting with 1 which is easy to do with a counter. The answer-validation algorithm uses a stack called deletestack. The contents of this stack are illustrated in table 2. The top of the stack is on the left in table 2.

Let us consider the kinds of tests that an answer-validation algorithm for a priority queue might perform. Suppose (i,k) is the answer to some deletemin operation. Further, suppose (i',k') was deleted in a previous deletemin operation. If the priority queue is correct then either (i,k)>(i',k') or (i',k') was deleted before (i,k) was inserted. This suggests that the time of insertion and deletion for elements should be recorded and the algorithm below does this. Unfortunately, if an algorithm compares an ordered pair which has been deleted against all previously deleted ordered pairs then the algorithm complexity is at least $O(m^2)$. To avoid this the deletestack is used. The deletestack was designed to allow many comparisons to be done implicitly and to reduce the complexity.

Algorithm for Answer Validation for Priority Queue

Input: sequence of $O(n)$ operations together with arguments and supposed answers for the priority-queue data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Array called *inserttime* indexed by item number. Array elements contain either "absent" or a time-stamp. Array called *keyvalue* indexed by item number. Array elements contain either "absent" or a key value. Initially, each element in these two arrays contains "absent". Stack of ordered triples called *deletestack*. Each ordered triple has the following form: first element is an item number, second element is a key value, and third element is a time-stamp. deletestack is initially empty.

In the first phase of the algorithm we process each operation as it appears serially using the following rules:

Let currenttime refer to the time-stamp of the operation being processed.

insert(i,k): If inserttime[i]≠"absent" then output "ill-formed" and stop. Otherwise, let inserttime[i] = currenttime and let keyvalue[i]=k.

deletemin (i,k): (where (i,k) is the supposed answer to the deletemin operation.) If inserttime[i]="absent" or keyvalue[i]≠k then output "ill-formed" and stop.

Otherwise, let (i',k') be the item number and key number of the triple on the top of deletestack (if there is one). Repeatedly pop the stack until (i,k)<(i',k') or until deletestack is empty.

If deletestack is empty then push the triple (i,k,currenttime) onto deletestack. Further, let insert-

time[i]="absent" and let keyvalue[i]="absent" and process the next priority queue operation.

If deletestack is non-empty then let the top element be (i',k',deletetime'). If inserttime[i]<deletetime' then output "incorrect" and stop. Otherwise, push the triple (i,k,currenttime) onto deletestack. Next, let inserttime[i]="absent" and let keyvalue[i]="absent" and process the next priority queue operation.

In the second phase of the algorithm we operate on the items which have been inserted but have never been deleted.

Scan the array inserttime and for each item number for which inserttime[i]≠"absent" construct an ordered triple (i,keyvalue[i],inserttime[i]). Call this set of ordered triples remainders.

Use a bucket sort to sort the triples in remainders by their time-stamps, i.e., the third element of the ordered triple.

Merge the triples in remainders together with the triples in deletestack so that they are all ordered by their time-stamps, i.e., the third element of the ordered triple.

Scan the combined triples to determine if there exist two triples which satisfy the following: inserttime[i]< deletetime' and (i,keyvalue[i])<(i',k'); where one triple is from remainders and has the form (i,keyvalue[i], inserttime[i]) and where the other triple is from deletestack and has the form (i',k',deletetime');

If these two triples exist then output "incorrect" and stop. Otherwise output "correct" and stop.

Theorem 7.1 *The algorithm for answer validation of the priority queue abstract data type is correct.*

Proof: Clearly the algorithm for answer validation always terminates. We must show that the algorithm outputs "correct" iff the operations together with arguments and supposed answers are correct. Because of space limitations we will only give a proof for the more difficult half of this iff statement. We shall use a proof by contradiction. Assume that the sequence of operations, arguments and supposed answers is considered correct by the algorithm but actually is incorrect. The use of the array inserttime and the symbol "absent" assures that no item is deleted when it is absent or inserted when it is already present. The use of the array keyvalue assures that items do not change keyvalue when they are present in the data type set. There is only one remaining way in which a sequence can be incorrect. This occurs when an ordered pair is deleted by a deletemin operation, however, it does not really have the smallest key value.

This means, there exist ordered pairs (i_1,k_1) and (i_2,k_2) such that $(i_1,k_1)>(i_2,k_2)$ and (i_1,k_1) is deleted while (i_2,k_2) is present in the data type set. In addition, we may specify that (i_1,k_1) is the largest ordered pair deleted while (i_2,k_2) is present. Let ins_1 be the time that i_1 was inserted and let del_1 be the time that i_1 was deleted. Let ins_2 be the time that i_2 was inserted and let del_2 be the time that i_2 was deleted (if it was deleted). There are two cases.

Case 1: Suppose that (i_2,k_2) is ultimately deleted. We know that $(i_1,k_1)>(i_2,k_2)$ by assumption. $del_2>del_1$ since item i_2 is deleted after item i_1. $ins_2<del_1$ since item i_2 was present when item i_1 was deleted.

Consider the situation when item i_2 is deleted with a deletemin operation. The ordered triple for item i_1 must appear in deletestack just before the processing of the i_2 deletion operation. This follows because the triple for item i_1 can only be removed from deletestack by a larger element and yet (i_1,k_1) refers to the largest ordered pair deleted while (i_2,k_2) was present. Now, since $(i_1,k_1)>(i_2,k_2)$ the ordered triple for item i_1 will remain in deletestack even after deletestack is popped during the processing of the deletemin operation for item i_2. Suppose the top of deletestack is (i_3,k_3,del_3) after the popping.

It is easy to show that the time-stamps on deletestack are monotonically ordered with the largest timestamp at the top. For this reason we know that $del_3 \geq del_1$. We noted earlier that $del_1>ins_2$. But if $ins_2<del_3$ then the algorithm outputs "incorrect" when it processes the deletemin operation. This contradicts our assumption that the sequence of operations, arguments and supposed answers was considered correct by the algorithm.

Case 2: Suppose the ordered pair (i_2,k_2) is never deleted. In the second phase of the algorithm the ordered triple (i_2,k_2,ins_2) is constructed and is compared against the ordered triples in deletestack.

The same argument that was used in case 1 above can be used to show that the test performed in the second phase of the algorithm would detect a problem and cause "incorrect" to be output. This contradicts our assumption that the sequence of operations, arguments and supposed answers was considered correct by the algorithm. Since both cases lead to a contradiction our proof is complete. ∎

Theorem 7.2 *The answer validation algorithm for priority queue has a time complexity of $O(n)$ for processing a sequence of $O(n)$ operations.*

Proof: We first analyze phase one of the algorithm. Note, there is a constant amount of work done for processing each single operation if we exclude the cost of popping the deletestack. Interestingly, popping the deletestack can take $O(n)$ time for the processing of a single operation. Luckily, the total amortized complexity for popping the deletestack while processing a sequence of $O(n)$ operations is still only $O(n)$. This

is true because each item which is inserted and later deleted is placed on deletestack and is popped at most once.

We now consider phase two. The cost of array scanning and constructing the triples is $O(n)$. The cost of the bucket sort is $O(n)$ and the cost of the merge is also $O(n)$. The final test can be implemented with a simple scan with a complexity of $O(n)$. Hence the overall complexity is $O(n)$ ∎

We have solved the answer-validation problem for abstract data structures that support the following set of operations: member, insert, delete, deletemin, min, deletemax, and max. The algorithm used to solve this problem is intricate but efficient. It requires only $O(n)$ time to process $O(n)$ operations. A detailed description of our solution, however, is beyond the scope of this version of the paper.

8 Conclusions

The results reported in this paper significantly generalize the applicability of the certification-trail technique. In our previously reported work on certification trails [11], we had to customize each algorithm application, but we have now developed trails appropriate to wide classes of algorithms. These certification trails are based on common data-structure operations such as those carried out using balanced binary trees and heaps. Any algorithm using these sets of operations can therefore employ the certification trail method to achieve software fault tolerance. To express the full generality of these ideas, we have provided constructions of trails for abstract data types such as priority queues and union-find structures. These trails are applicable to any data-structure implementation of the abstract data type. These ideas lead naturally to monitors for data-structure operations. We are currently working on an experimental evaluation of the approach and initial results are promising.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[4] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[5] Fredman, M. L., and Saks, M. E., "The cell probe complexity of dynamic data structures," *Proc. 21st ACM Symp. on Theo. Comp. 1989*, pp. 109-122, 2, 1986.

[6] Gabow, H. N., and Tarjan, R. E., "A linear-time algorithm for a special case of disjoint set union," *J. of Comp. and Sys. Sci.*, 30(2), pp. 209-221, 1985.

[7] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[8] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[9] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[10] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[11] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[12] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Department of Computer Science Technical Report JHU 90/17*, Johns Hopkins University, Baltimore, Maryland, 1990.

[13] Tarjan, R. E., "Efficiency of a good but not linear set union algorithm," *J. ACM*, 22(2), pp. 215-225, 1975.

[14] Tarjan, R. E., "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. of Comp. and Sys. Sci.*, 18(2), pp. 110-127, 1979.

[15] Tarjan, R. E., and Leeuwen, J. van, "Worst-case analysis of set union algorithms," *J. ACM*, 31(2), pp. 245-281, 1984.

[16] Taylor, D., "Error Models for robust data structures," *Dig. 20th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 416-422, 1990 June 26-28.

[17] Williams, J. W. J, "Algorithm 232 (heapsort)," *Commun. of ACM*, vol.7, pp. 347-348, 1964.