Solving Large Problems Quickly: Progress in 2001-2003

Todd C. Mowry Angela Demke Brown
Christopher B. Colohan J. Gregory Steffan Antonia Zhai

April 2004
CMU-CS-04-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This document describes the progress we have made and the lessons we have learned in 2001 through 2003 under the NASA grant entitled "Solving Important Problems Faster". The long-term goal of this research is to accelerate large, irregular scientific applications which have enormous data sets and which are difficult to parallelize. To accomplish this goal, we are exploring two complementary techniques: (i) using compiler-inserted *prefetching* to automatically hide the I/O latency of accessing these large data sets from disk; and (ii) using *thread-level data speculation* to enable the optimistic parallelization of applications despite uncertainty as to whether data dependences exist between the resulting threads which would normally make them unsafe to execute in parallel. Overall, we made significant progress in 2001 through 2003, and the project has gone well.

This research is supported by a grant from NASA.

Keywords: B.3.2 Cache Memories, C.1.2 Multiple Data Stream Architectures (Parallel Processors), C.4 Performance of Systems, D.3.4 Compilers

Chapter 1

Introduction

A number of important scientific and engineering applications suffer from the following two limitations. First, the largest problem size that one can solve in practice is constrained by the amount of physical memory in the system. Although one could (in theory) attempt to execute an application with a data set that is too large to fit in main memory (commonly referred to as an "out-of-core" application) by simply relying on a paged virtual memory system to migrate the data between memory and disk transparently, the resulting performance is typically unacceptable. Programmers are thus faced with the formidable task of restructuring their code to use explicit I/O calls for the sake of achieving reasonable performance. Second, the task of decomposing applications which use sophisticated data structures (involving pointers and other forms of indirection) into parallel threads is quite onerous since their unpredictable memory access patterns make it difficult (and in many cases impossible) to statically prove that potential threads are in fact independent. This latter problem is particularly troublesome for large "legacy" codes where many years of effort have already been invested in simply getting a sequential version of the program to work correctly.

The goal of this research project is to overcome both of these limitations using the following two techniques. First, to achieve high performance on out-of-core data sets, we explore a fully-automatic scheme for prefetching I/O whereby the operating system and the compiler cooperate to combine the advantages of both explicit I/O (i.e. high performance) and paged virtual memory (i.e. portability and no burden on the programmer) without suffering from the disadvantages of either approach. Second, to overcome the problem of ambiguous data dependences between potential parallel threads, we explore a new mechanism called Thread-Level Speculation (TLS) which enables either the programmer or the compiler to optimistically speculate that no dependences exist between the threads and run them in parallel.

1.1 Progress Toward Milestones

In our proposal, we set forth the following eight technical milestones. (The ninth milestone was to complete and deliver this technical report, so we do not list it below.)

1. Implement our new compiler algorithm for scheduling prefetches within the SUIF compiler framework, and evaluate this algorithm on the NAS Parallel benchmarks, as well as on other applications such as Overflow and MM5. In our experiments to date, we have often found prefetch scheduling to be the major limitation in hiding I/O latency effectively. We expect the new algorithm to deliver significant performance improvements in many cases, and to be more adaptable to changes in the run-time environment.

2. Evaluate and tune our Linux implementation for supporting prefetch and release requests. In the previous year, we were able to implement the full functionality for our system within Linux, however, we have not investigated the performance of this implementation on a multi-disk machine. Given the cost-effectiveness of Linux machines, it is extremely desirable to be able to support large-scale applications in such an environment.

3. Evaluate the performance impact of our techniques on Overflow, the MM5 Weather Modeling code, and perhaps other applications that are of interest to NASA. We have recently obtained an out-of-core data set for Overflow, and will investigate how our techniques can be applied to improve the performance of this application.

4. Pursue opportunities for transferring the IRIX operating system support for prefetch and release operations back to SGI, with the goal of seeing these features become part of a production operating system release. Similarly, make the Linux implementation publicly available, once it has been fully evaluated and major performance issues have been corrected.

5. Complete our evaluation of the best system support for achieving high performance using thread-level speculation. Actively disseminate this information to the major high-performance microprocessor companies and help them choose the right support for TLS in future designs.

6. Complete our implementation of a compiler that exploits TLS to automatically convert irregular sequential codes into parallel codes. Evaluate its effectiveness and improve upon its design. Make the source code for this compiler available to NASA.

7. Develop new compiler techniques for choosing the appropriate parallel thread boundaries within sequential codes for the sake of achieving high program coverage and maximizing overall program speedup using TLS. Maximize the scope of what can be parallelized using TLS.

8. Develop new compiler techniques scheduling techniques to minimize lost performance due to frequently forwarded data values across threads under TLS. (We have observed that this is an important performance bottleneck.)

Regarding our progress toward these milestones, we far exceeded the milestones related to TLS (milestones 5 through 8). We did not reach all of our milestones for I/O prefetching, unfortunately, but we still made very good progress in that area as well. Here is a quick summary of the progress that we made toward each milestone (further details are presented later in this paper).

Milestone 1: We did implement and evaluate our algorithm, as described in Chapter 2. The performance benefit was not as large as we had hoped, and we are continuing to investigate whether the algorithm can be further improved.

Milestone 2: We have completed this milestone. Chapter 3 describes our experiments using the Linux implementation of I/O prefetching.

Milestone 3: Unfortunately we did not make as much progress toward this milestone as we had hoped. We attempted to apply our techniques to Overflow, but we were unable to get it to work properly on our machines. We are still working on applying our techniques to MM5, and we will present those results in Angela Demke Brown's PhD thesis document once they are available.

Milestone 4: Since we proposed this milestone of transferring our technology to a production version of IRIX, it became irrelevant, unfortunately, because SGI stopped developing the IRIX operating system. Instead, SGI now runs Linux on their systems. Since we already have a version of our prefetching support for Linux and we can publicly release it ourselves, we do not need to work with SGI to transfer this technology.

Milestone 5: We completed this milestone. Our design is described in detail in Chapters 4 and 5. We have given talks on this design to microprocessor groups at Intel and IBM, and they are quite interested in our approach. We expect that TLS support will start appearing in commercial microprocessors sometime in the near future.

Milestone 6: We completed this milestone, as described in detail in Chapters 6 and 7.

Milestone 7: We completed this milestone, as described in detail in Chapter 8.

Milestone 8: We completed this milestone, as described in detail in Chapters 6 and 7.

Overall, we are pleased with the outcome of the project. In the following sections, we discuss our research progress in more detail.

PART I: I/O Prefetching

Chapter 2

I/O Prefetching on IRIX

In previous publications, we have described in detail our compiler algorithm for prefetching and releasing and showed it to be a promising approach for handling the memory demands of out-of-core numeric benchmarks when combined with run-time and operating system support [56, 10]. We also showed, however, that there were numerous situations where the code transformations made at compile-time could not be adequately adapted to deal with dynamic run-time conditions.

In this chapter, we report on our experience with using a new compiler algorithm for scheduling prefetch operations and show that this algorithm is able to vastly reduce the number of late prefetches. That is, most prefetch requests are issued early enough to complete before the data is needed when the new algorithm is used. We begin in Section 2.1 with a discussion of the scheduling challenges that our new algorithm needs to address. We then develop our new algorithm incrementally in Section 2.2. We also evaluate the impact of the nested pipelining algorithm on the performance of FFTPDE, one of the NAS Parallel benchmarks.

2.1 Scheduling challenges

Our compiler algorithm for inserting prefetch requests into application source code uses *software pipelining* to ensure that the request for data is issued early enough to hide the latency of a disk fetch. If we can construct an effective pipeline of prefetch requests and data accesses across a set of loops, we can substantially improve performance by avoiding stalls due to page faults.

Software pipelining is a technique commonly applied to expose instruction-level parallelism in loops. Our usage of the technique was adapted from its application to the problem of scheduling prefetches for cache misses in looping codes [57]. In both cases the latencies that need to be hidden by the pipeline are known at compile-time and are small relative to the size of the loop being pipelined. Under these conditions, it is reasonable to compile for a fixed latency when constructing the pipeline and to consider only the innermost loop nest. When software pipelining is used to schedule prefetches for page faults in out-of-core looping codes, however, one or both of these properties may not hold.

Our initial compiler algorithm for I/O prefetching used a fixed latency estimate and attempted to address the problem of small inner loops by introducing a simple heuristic: construct the pipeline across the innermost loop that accesses a sufficient amount of data [56]. This technique can help to select a loop that is large enough for a given latency, but only if the loop bounds are known at compile-time so that the amount of data accessed can be calculated. When loop bounds are symbolic, choosing the right loop for pipelining remains a problem. The combination of nested loops and large latencies creates a third problem for software pipelining of I/O prefetch requests: the pipeline is repeatedly filled and drained on each iteration of the surrounding loop, and the time to initialize the pipeline may be large compared to the time spent in the steady state. We now elaborate on the problems of finding the right prefetch distance and dealing with nested loops.

2.1.1 Variations in Prefetch Distance

The first step in the software pipelining algorithm (for both the original I/O prefetching version, and our new variations) is to determine the *prefetch distance*. This distance is expressed as a number of loop iterations, and is calculated using

an estimate of the I/O latency, and the shortest path through the loop body. If our calculated prefetch distance is too large, then we will increase the memory pressure since more pages are needed to hold prefetched data. As a result, evictions may occur earlier than necessary. Over-estimating the prefetch distance by a small amount is not a serious problem, however, because main memory has plenty of capacity to buffer the prefetch requests and we can use the release hints to make good replacement decisions. If, on the other hand, we underestimate the prefetch distance, then we will be unable to hide all of the I/O latency. There is no way for the run-time layer to compensate for a late prefetch request by issuing it earlier—by the time the run-time layer sees the request, it is already too late. There are two basic reasons that we may be unable to estimate the prefetch distance accurately at compile-time: the estimate of the time to execute the loop body may be inaccurate, and/or the estimate of the I/O latency may be inaccurate.

Causes of Inaccurate Estimates

We estimate the time to execute one iteration of the loop body by counting SUIF instructions in the shortest path through the loop. At this point in the compilation process, the instructions are in an intermediate format, so we do not know the actual machine instructions that will be executed at run-time. Some of these instructions may be removed completely by later optimization passes. We make an additional simplifying assumption that each SUIF instruction will execute in one cycle. These inaccuracies could cause the loop to execute faster than expected, resulting in late prefetches which cannot hide all the I/O latency. Even if we knew the actual cycle count for the shortest path, we would still have a problem with calculating the prefetch distance. The actual path through the loop taken at run-time may be many times longer than the worst-case shortest path; this would result in prefetches being issued earlier than necessary.

The I/O latency estimate is a compile-time parameter which we have chosen based on measurements of our target systems. It is expressed in terms of processor cycles, so both the disk speed and the clock speed of the target system are implicitly included in the estimate. Even on a single, uncontended system, however, the worst-case time to read a page from disk could be about twice as long as the average case depending on the disk seek time and rotational latency required to locate the data. Using an average measured value means that some pages may be requested too early, while others will be requested too late. We could conservatively double the average page fetch time, relying again on the large buffering capacity of main memory, and the release hints to cope with the inflated prefetch distance, but there is another more serious problem. We do not, in general, want to recompile the applications for each target system.

Consider, for example, general-purpose computers based on the Intel Pentium IV architecture. Currently, the same binary program could execute on processors that differ by more than 1 GHz in clock speed. At the same time, a particular system built around these processors could include anything from 5400-7200 IDE disks to 10K-15K SCSI disks. It is highly-undesirable (if not completely unreasonable) to have to compile a unique binary for each configuration on which the application will execute, just so the target latency parameter can be specified.

Solution: Variable Prefetch Distances

Rather than compile for a fixed latency estimate and a statically-estimated dynamic loop execution time, we would prefer to express the prefetch distance as a variable at compile-time and generate code to calculate it dynamically based on actual behavior. There are numerous options for obtaining the actual latency at run time. For instance:

- Measure the actual time for a page fetch (this could be done once for a given system and stored somewhere under the /proc filesystem where all interested processes could read it). Time to execute a loop body can start with a static estimate of loop length but use profiling information (such as cycle counters) to measure the actual time to execute the loop body. With these two parameters obtained at run time, we can now calculate the prefetch distance as the latency divided by the loop execution time as before.

- Track the percentage of late prefetches in a loop and dynamically increase the prefetch distance until the late fraction becomes acceptably small. It would likely be useful in this case to include additional inforation, such as disk queue length, to avoid continually increasing the prefetch distance in a system that is bandwidth-limited.

Other options are also available, but an important question is whether or not using a variable latency, and adding code to calculate the prefetch distances at run-time, rather than calculating these values statically at compile-time, introduces substantial run-time overhead.

To address this question, we modified our compiler to generate code that calculates the prefetch distances dynamically at run-time. The latency estimate is simply entered from an input file, or the command-line, since we are

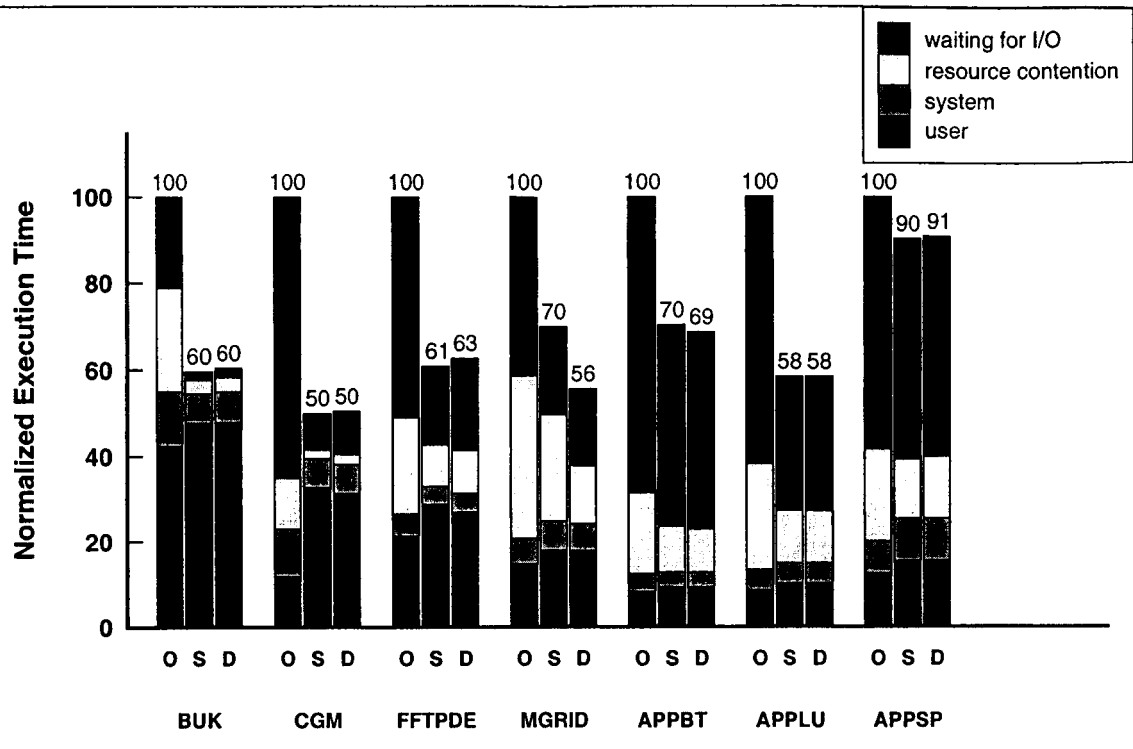


Figure 2.1: Effect of calculating prefetch distances at run-time. Bars labeled "O" are the original, non-prefetching version; bars labeled "S" use a fixed compile-time latency; bars labeled "D" use a dynamic latency value obtained at run-time.

primarily interested in evaluating the overhead of performing these calculations at run-time, not in exploring how to measure latency in a running system. We also added code to calculate the length of a loop body at run-time, allowing the actual bounds of internal loops to be used, rather than assuming a worst-case execution of a single iteration. The results of this experiment are shown in Figure 2.1. We have omitted EMBAR from this study, as it has only a single interesting loop, with only a single, sequential array access. Even the most straightforward prefetching strategy is able to hide the latency for this benchmark.

In most cases, the differences between using a static latency value at compile-time, and a dynamic value obtained at run-time is negligible. A notable exception is MGRID which benefits substantially from the run-time calculations. From the graph, we can see the the improvement comes from a reduction in the "resource contention" component. MGRID has an interesting access pattern where the loop bounds in a single function change across calls to that function. In this situation, it is impossible to make a reasonable estimate of the length of the loop bodies at compile-time, and the conservative estimate of a single iteration leads to an excessively large prefetch distance on most calls. These extra prefetches create contention in the operating system as they are handled. When we can calculate the prefetch distance more accurately using the actual run-time parameters, these excess prefetches are reduced and the contention over locks in the memory system drops substantially.

Overall, the potential benefit of calculating prefetch distances at run-time (as illustrated by the improvements in MGRID), and the increased flexibility (due to not needed a recompilation if the expected latency changes) make this an extremely useful feature to incorporate into our compiler, independent of any changes to the actual scheduling algorithms. We now consider several alternatives for further improving the performance of perfetching on benchmarks with multi-dimensional nested loops such as FFTPDE, MGRID, APPBT, APPLU, and APPSP.

2.1.2 Nested loops

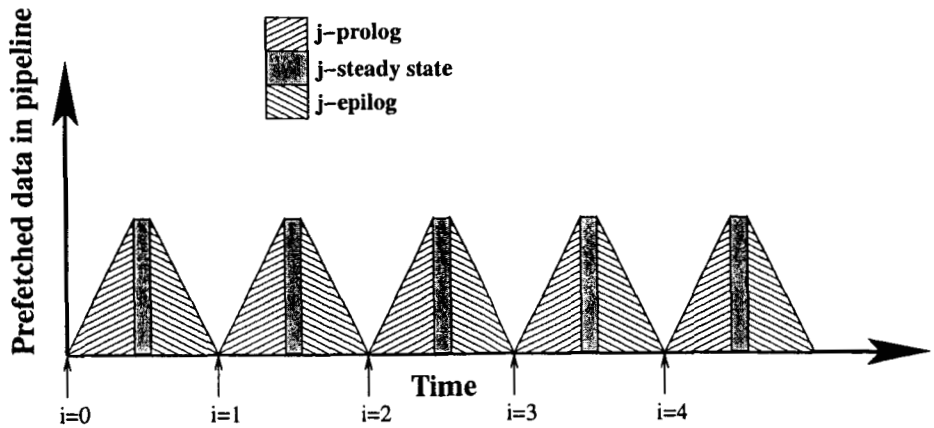
Even under the ideal conditions for nested loops (perfectly nested loops with compile-time constant bounds), we can lose a lot of opportunities for latency-hiding. The code in Figure 2.2(a) shows a simple two-dimensional loop containing an access to a two-dimensional array, and in Figure 2.2(b) we show how we would construct a software-

```
for (i = 0; i < Ni; i++)
   for (j = 0; j < Nj; j++)
      access(a[i][j]);
```

(a) Simple two–dimensional array accesses

```
for (i = 0; i < Ni; i++) {
prolog  for (jprolog = 0; jprolog < min(d, Nj); jprolog++)
           prefetch(&a[i][jprolog]);
steady  j_sw_pipe_upperbound = max(0, Nj-d);
state   for (j = 0; j < j_sw_pipe_upperbound; j++) {
           prefetch(&a[i][j+d]);
           access(a[i][j]);
        }
epilog  for (j = j_sw_pipe_upperbound; j < Nj; j++)
           access(a[i][j]);
}
```

(b) Software pipelining prefetches around inner loop
(The calculation of the prefetch distance, d, is not shown)



(c) Effect of repeatedly filling and draining the pipeline

Figure 2.2: Adding prefetches for two-dimensional array accesses

pipeline around the inner loop (we will refer to this as the *j-loop*) to prefetch the data accesses. For simplicity, we do not show the effect of the strip mining optimization which reduces the frequency of prefetch requests. Suppose that the number of iterations of the j-loop, `Nj`, is only slightly larger than the calculated prefetch distance, `d`. In this case, we will execute very few iterations in the steady-state. Nearly all of the prefetches will be issued in the prolog section to fill the pipeline, and nearly all of the data accesses will occur in the epilog section to drain the pipeline. Further, because the j-loop is nested inside the outer i-loop, the cycle of filling and draining the pipeline is repeated on each iteration of the outer loop. Figure 2.2(c) depicts the effect of this repeated fill and drain cycle in terms of the number of prefetched pages in the pipeline for the case where `Nj` is slightly larger than `d`.

In general, we do not know whether `Nj` will be larger than `d` or not at compile-time. If the j-loop is not large enough to hide all the latency, then the steady-state is never reached at all. In this case, the i-loop would have been a better choice for building our pipeline. On the other hand, if `Nj` is much larger than `d`, then the cost of filling the pipeline will be small relative to the time spent in the steady-state and the i-loop would be a poor choice for pipelining. Clearly, there is no single choice of loop that will be right for all circumstances. Instead, we should take all the loop levels into account and avoid draining the pipeline unnecessarily.

2.2 Developing the Continuous Software Pipelining Algorithm

In this section we consider a sequence of techniques that address various issues with multi-dimensional loops, leading up to the full *continuous software pipelining* algorithm. Throughout, we will focus on the two-dimensional case, although the solutions generalize to higher dimensions.

2.2.1 Nested pipelines

Consider again the case of a two-dimensional array being accessed within a two-dimensional set of loops, as shown in Figure 2.2(a). When we access an element, `a[i][j]`, in the steady-state of our software pipeline, we want to prefetch the element that will be accessed d iterations of the inner loop in the future. If we only consider the inner loop, then we could prefetch `a[i][j+d]`, as shown in Figure 2.2(b). Depending on the length of the inner loop, relative to the prefetch distance, this strategy may or may not be successful at hiding the latency. However, there may be enough work to hide the latency if we take the surrounding loop into account as well by calculating the prefetch address as a function of both loop indices. To prefetch d iterations of the inner loop ahead, we will need $d \text{ div } N_j$ iterations of the outer loop, leaving $d \bmod N_j$ iterations of the inner loop. The offsets for each loop index are given below:

$$d_i = d \text{ div } N_j \tag{2.1}$$

$$d_j = d \bmod N_j \tag{2.2}$$

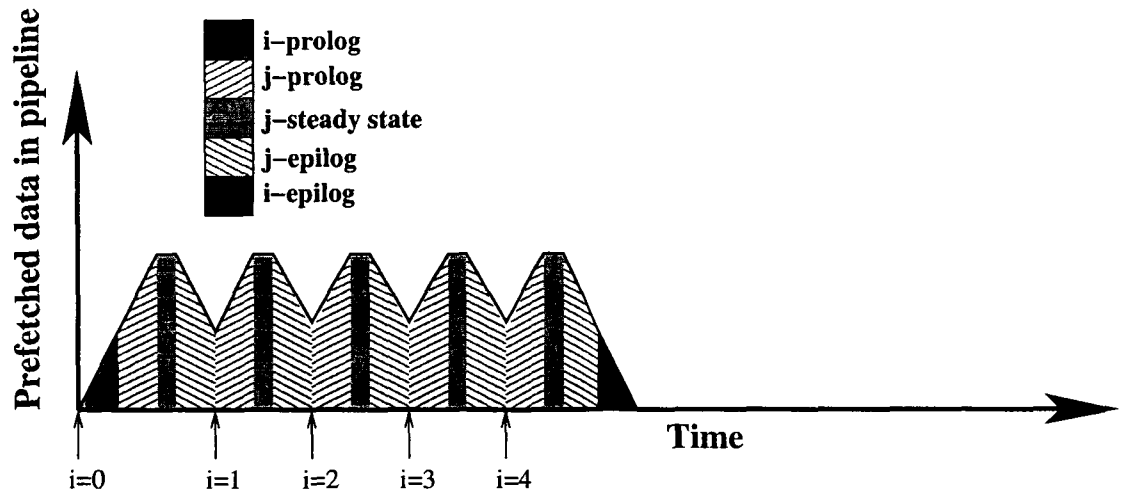
The prefetch address (for the reference of Figure 2.2(a)) is then simply `a[i+`d_i`][j+`d_j`]`. Using these offsets for the prefetch address, and constructing the pipeline around the inner loop so that the first d accesses are prefetched in the prolog gives the code shown in Figure 2.2(c). Unfortunately, this code still isn't quite right. For every iteration of the outer loop after the first iteration, in the prolog we repeat the prefetch of some of the same elements that we prefetched on the previous iteration. This is caused by the fact that the first time we see the loop we need to issue a large number of prefetches to fill the pipeline, but on subsequent iterations of the outer loop, we only need to re-fill the portion that was drained during the epilog of the inner loop. The logical solution is to pull out the portion of the prolog that issues prefetches for the outer loop offset (d_i), placing it before the first iteration of the outer loop. Also, it would be useful to have an epilog for the outer loop as well, to avoid prefetching more than necessary along the outer dimension. The resulting code has the appearance of *nested software pipelines*, as shown in Figure 2.3(a). Around each loop nest we have constructed a pipeline to handle address offsets involving that loop's index variable. Viewed in terms of the effect on prefetched data, the inner epilog only partially drains the pipeline, allowing more of the latency to be hidden, as depicted in Figure 2.3(b). First, prefetches are issued from the outer prolog, then we have a repeated pattern of filling the inner pipeline, executing the inner steady state, and draining the inner pipeline. The outer pipeline is not drained until we get to the end of the pair of loops, which is the real end of the data access.

The technique of nested pipelines can be applied to an arbitrary number of dimensions. For n dimensions, we will have $n + 1$ copies of the original loop body. Each loop nest adds one extra copy in its epilog, plus there is one

```
i-prolog
    for (ip = 0; ip < min(di, Ni); ip++)
       for (j = 0; j < Nj; j++)
          prefetch(&a[ip][j]);

    i_swp_ub = max(0, Ni-di);
    for (i = 0; i < i_swp_ub; i++) {
j-prolog
       for (jp = 0; jp < min(dj, Nj); jp++)
          prefetch(&a[i+di][jp]);
i-steady state
       j_swp_ub = max(0, Nj-dj);
j-steady state
       for (j = 0; j < j_swp_ub; j++) {
          prefetch(&a[i+di][j+dj]);
          access(a[i][j]);
       }
j-epilog
       for (j = j_swp_ub; j < Nj; j++)
          access(a[i][j]);
    }
i-epilog
    for (i = i_swp_ub; i < Ni; i++)
       for (j = 0; j < Nj; j++)
          access(a[i][j]);
```

(a) Sample code for nested software pipelines



(b) Effect of nested pipelining on prefetched data

Figure 2.3: Nested software pipelines for two-dimensional array prefetches

additional copy in the innermost steady-state. Although it is possible to construct examples involving any number of loop nests, all of the concepts we need to illustrate can be shown with only two levels, so we will focus on the two-dimensional problem in the remainder of this discussion.

At this point we have solved the problem of generating the correct prefetch address when the distance required to hide the I/O latency spans an arbitrary number of loop nests. However, we can still do a better of job of scheduling the prefetches. Figure 2.3(b) shows a general view of the effect of nesting the software pipelines, but the actual effectiveness depends on the amount of latency that we are attempting to hide with each of the nested pipelines. For example, if $N_j > d$ when we calculate the depth d_i for the outer pipeline using Equation 2.2, then d_i will be zero and the situation will be identical to Figure 2.2(c). We are only better off multiple loop nests are actually needed to hide all the latency, but we will typically not be able to know if this is the case or not until run-time.

To solve this problem, we need to start re-filling the pipeline with data from the prolog of the next iteration as we drain out the data in the current epilog. In effect, we have to merge the epilogs and prologs together.

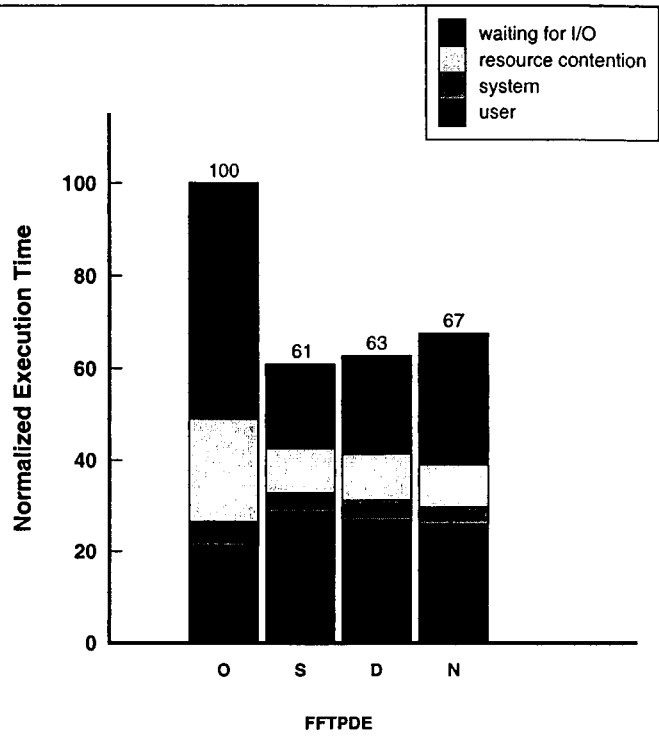


Figure 2.4: Effect of nested pipelining on FFTPDE. ("O" = Original, non-prefetching version; "S" = Static, compile-time latency; "D" = dynamic latency value obtained at run-time; "N" = nested pipelining with dynamic latency)

Evaluation of Nested Pipelining

We have implemented the algorithm for nested software pipelining as part of the scheduling phase of our prefetching pass for the SUIF compiler. To illustrate the effect of nested pipelining, we focus on the FFTPDE benchmark.

Figure 2.4 shows the overall effect on execution time, with the non-prefetching version (O), the original static prefetching algorithm (S), and the original algorithm with dynamic latency (D) shown for comparison with the nested pipelining algorithm (N). Looking at the components of the "N" bar, we see that nested pipelining has slightly better performance in terms of user execution time, time spent executing system code, and contention in the operating system. These improvements, however, come at the cost of increased I/O stall time leading to a slight overall slowdown as compared to the previous prefetch scheduling algorithm. To understand the reasons for these results, we take a closer look at the prefetch coverage (how effectively the original page faults are prefetched) and the components of the I/O stall time.

Figure 2.5 illustrates what happens to the page faults of the original program when the three different prefetching approaches are used. The "non-prefetched fault" category represents references which are not prefetched at all. The "prefetched hit" category represents references which incurred a page fault in the original program, but do not fault in the prefetching version (that is, the prefetch has completed successfully before the reference occurs). The "prefetched fault" category represents those references that were prefetched too late; only some of the page fault latency has been hidden. From this figure, it is clear that calculating the prefetch distance at run-time has no effect on the coverage for FFTPDE. The nested pipelining algorithm is more successful at hiding the latency when it issues prefetches (note the smaller "prefetched fault" component), however, it has more than twice as many references that are not prefetched at all. This result occurs because the original scheduling algorithm often prefetches large blocks of pages in the prolog sections, even when those pages are not going to be used in the relevant loop nest. In some cases, this excess prefetched data is retained in memory and is available for a future reference (although not the one intended when the data was prefetched!). Essentially by accident, the original algorithm gains a better coverage factor. Considering the number of prefetch requests that are issued to the OS, the original algorithm sends roughly 120,000 more than the nested pipelining algorithm, and is able to eliminate roughly 50,000 more page faults as a result. The other 70,000 requests, however, place extra strain on the memory system, increase contention, and waste disk bandwidth. It would be better

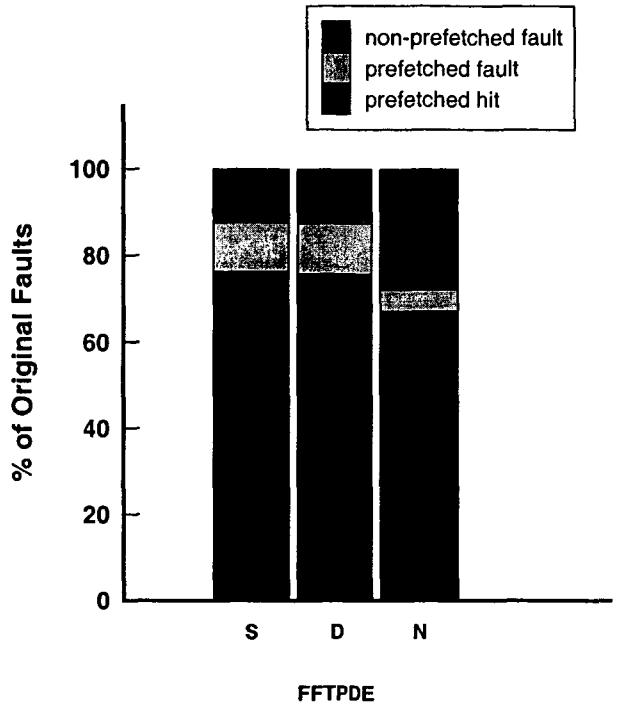


Figure 2.5: Comparing prefetch coverage for FFTPDE. ("S" = Static, compile-time latency; "D" = dynamic latency value obtained at run-time; "N" = nested pipelining with dynamic latency)

to focus on the analysis phase to determine why some references are not prefetched under either scheduling algorithm, and then apply the lower-overhead nested pipelining algorithm.

As further evidence that the nested pipelining algorithm is effectively meeting its goals of issuing prefetches early enough to hide all the I/O latency, we break the "waiting for I/O" component of Figure 2.4 into two categories. The first is time spent waiting for a page that had already been prefetched. This represents "late" prefetches, and corresponds to the cost of the "prefetched miss" category in Figure 2.5. The second is time spent waiting for pages that were not prefetched at all, and represents the cost of the "non-prefetched fault" category in Figure 2.5. Figure 2.6 shows these results for the original, static (S) scheduling, the original scheduling with dynamic latencies (D), and the nested pipelining algorithm (N). From this figure, we can see that only about 2% of the remaining I/O stall time for FFTPDE is due to prefetches that were issued too late. This is a substantial contrast to the previous scheduling algorithm where nearly 60% of the stall time was due to late prefetches.

Although there is potential for the nested pipelining algorithm to be ineffective, depending on the specific loop bounds and latencies seen at run-time, we see that in FFTPDE this does not occur. The remaining sections of this chapter discuss algorithms for merging the epilog of one iteration with the prolog of the following one, to ensure the software pipeline of prefetches remains continuously filled. This technique is intended to eliminate the remaining latency that cannot be hidden by nested pipelining when outer loop pipelines drain and refill. We have not evaluated these algorithms on FFTPDE because there is very little latency left after applying nested pipelining. To further improve the performance of FFTPDE we need to instead focus on the references that are not being prefetched at all, rather than the scheduling algorithm.

2.2.2 Merging prologs and epilogs

Figure 2.7 depicts the effect of merging the epilog of one iteration with the prolog of the following iteration. The dashed lines correspond to the epilog and prologs from the nested pipelining case that have now been merged together, keeping the total number of prefetches in the pipeline steady. There is now a single "fill" stage when we are unable to hide all the I/O latency, but for most of the execution we are able to stay in the steady-state.

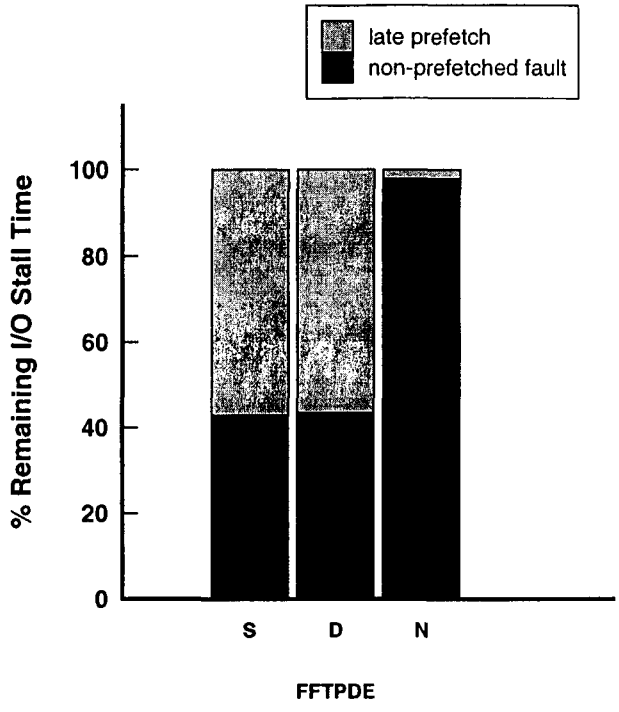


Figure 2.6: Comparing cost of late prefetches vs. non-prefetched references for FFTPDE. ("S" = Static, compile-time latency; "D" = dynamic latency value obtained at run-time; "N" = nested pipelining with dynamic latency)

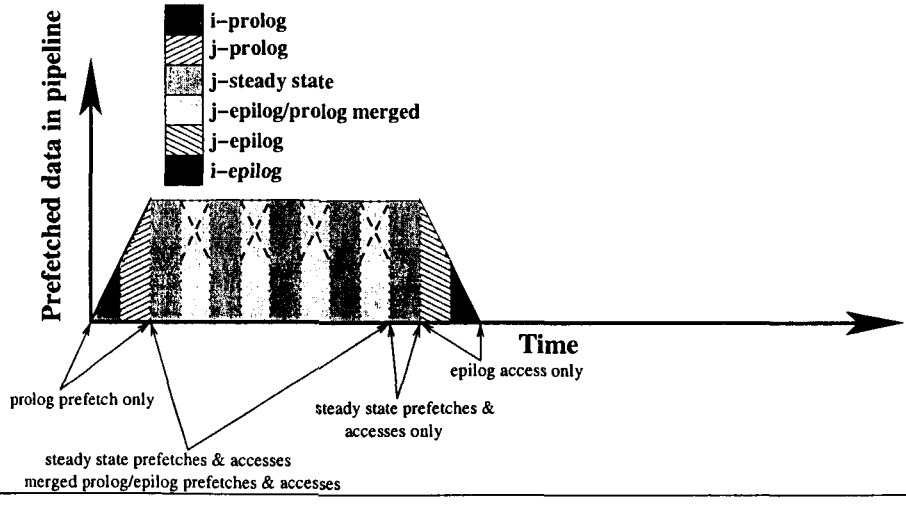


Figure 2.7: Continuous software pipelines for two-dimensional array prefetches

To build a software pipeline that will have the behavior shown in Figure 2.7 we need to do three things. First, in the inner epilog we need to insert the prefetches that would be issued in the inner prolog of the next outer iteration. To do so, we simply need to calculate the address that would be used for that prolog prefetch, where the outer loop index variable has increased by the loop step and the inner index variable has been reset to its lower bound. For this example, the address to prefetch is `a[i+di+1][j+dj-Nj]`. Second, since the inner prolog and epilog are being merged, we can now pull the first instance of the inner prolog out of the loops, making it part of the outer prolog. Third, we need to identify the point where we no longer need to merge the inner prolog and epilog by peeling off the last iteration of the

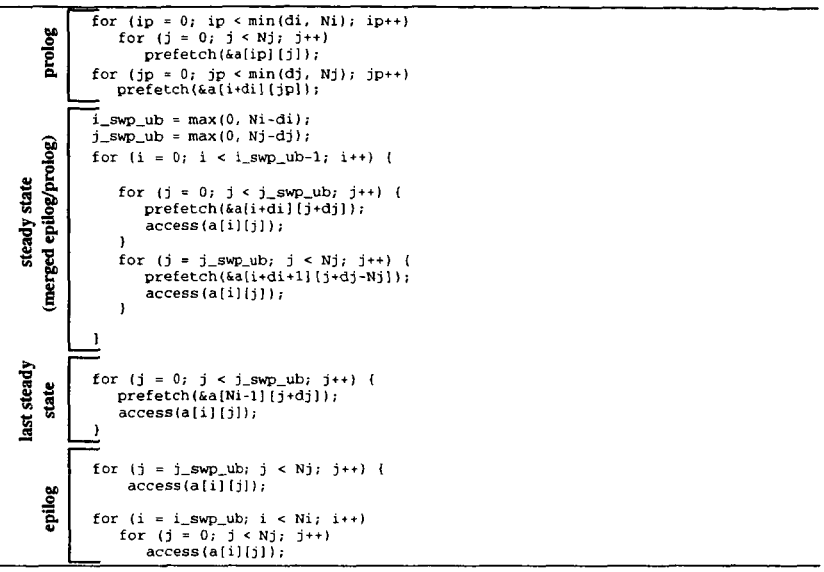


Figure 2.8: Continuous pipelining: after initial prolog, pipeline stays filled until no more prefetches are needed.

outer steady state. In this iteration, we still want to issue the inner steady-state prefetches, but we don't have another prolog to merge into the epilog.

Taking these three steps yields the continuous software pipeline shown in Figure 2.8. This code has all of the properties we want for I/O prefetching: the prefetch addresses are calculated correctly to hide large latencies, and the software pipeline remains filled as long as necessary. Issuing prefetches is not the only issue, however. Note that there are now five copies of the original loop body. In general, the strategy outlined here will lead to $O(2^n)$ copies of the original loop body, for n levels of loops. Clearly, this leads to an unacceptable expansion in code size for even moderately deep nestings.

2.2.3 Reducing code expansion

It is tempting to address the problem by ignoring the final steady state prefetches and going straight to the epilog when we no longer have inner epilogs and prologs to merge together. This eliminates the need to peel an iteration of the outer loop, and reduces the number of copies to what we had for the basic nested pipelines. Unfortunately, if we do not know the loop bounds, we can't estimate how important those final steady state prefetches really are. For example, if `Ni` is only 2, and `Nj` is extremely large, roughly half of the prefetches should be issued in the last steady state. By ignoring this possibility, we greatly reduce our opportunities to hide I/O latency. Fortunately, there is a better solution. All the copies of the original loop body occur in one of only two situations: either the original body by itself, or the original body with a prefetch operation added. The only differences are the upper and lower bounds on the loops surrounding the body, and the offsets used in the prefetch address calculation. By identifying the points where these values need to change and inserting code to calculate them appropriately, we only need a single copy of the body with the prefetch operation. Similarly, if we can identify the point where prefetching should stop, we only need a single copy of the epilog code.

Our solution is to "wrap" each inner loop in a new loop that executes at most twice. Before the first iteration of the wrapper we insert code to set the bounds on the "wrapped" loop for the steady state condition. At the end of the wrapper, we insert code to set bounds and offsets for the merged epilog/prolog situation. An example for the two-dimensional case is shown in Figure 2.9. This approach introduces some additional overhead, in the form of the extra loop and the code to update bounds and offsets.

We believe that these techniques will generate code that can adapt more effectively to a wide range of input conditions and latencies, however, for the benchmarks that we have, these advanced adaptivity features have not been necessary.

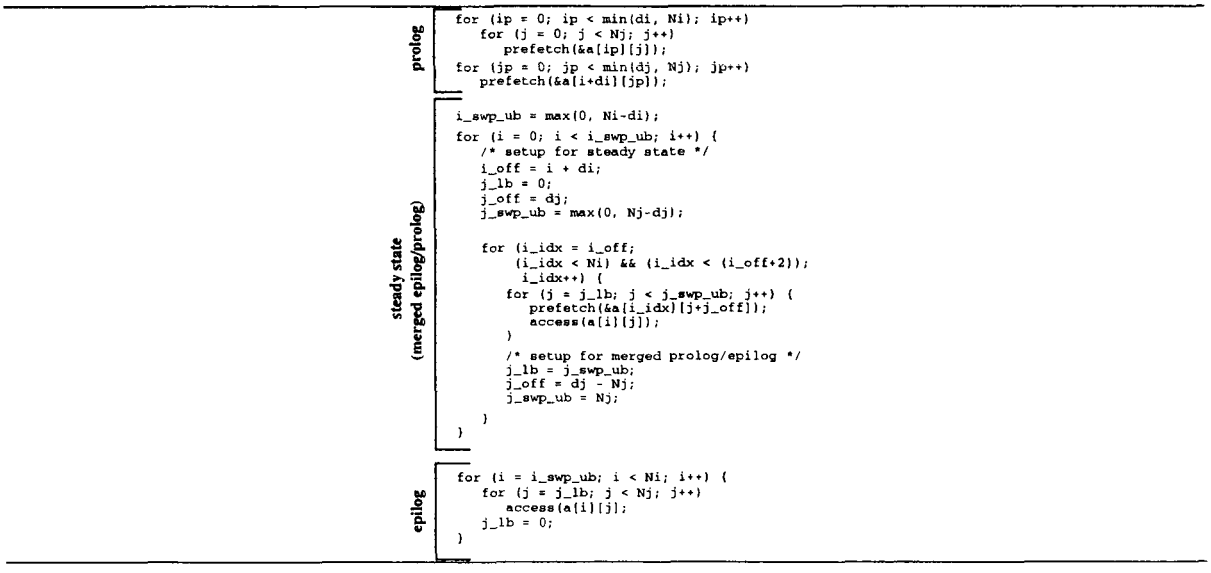


Figure 2.9: Wrapped pipeline: only two copies of the original loop body are needed to cover all situations.

Chapter 3

I/O Prefetching on Linux

3.1 System Architecture

Our overall architecture [56, 9] consists of three major components: a compiler, operating system support for prefetch and release calls, and a user-level library that adjusts the application's behaviour to run-time conditions. Each of these components is described in brief in the following subsections.

3.1.1 Compiler

The compiler has to analyze the code and determine its data access patterns. As mentioned earlier, this problem is hard in general, but can be done quite easily and with a high degree of success for regular code. This characteristics is present in most scientific applications that need to process large data sets. By regular code, we mean that loops in these programs, in which most of the computation and data access takes place, have simple data-flow, i.e. they must be preferably single entry, single exit and if there is any nesting of loops then any non-loop instructions are part of all loop bodies.

The compiler uses *locality analysis* to predict when misses (i.e. page faults) are likely to occur. This is done by determining when a new page of virtual memory is accessed (i.e., the part of the array being accessed is on a different page), and if that page is likely to have been accessed in the near past. When potentially faulting instances are identified, the compiler first isolates them through *loop splitting* techniques and then uses *software pipelining* to schedule these prefetches early enough.

After its analysis is over, the compiler needs to pass this information to the operating system by instrumenting the code. One alternative is to pass a summary of future access patterns to the operating system through a single call at the start of execution. However, this approach is undesirable, since the access patterns in real applications often depend on the dynamic behavior that can only be determined at run time. Also, expressing the summary information concisely would greatly complicate the compiler/operating system interface. Another disadvantage of this approach is that it pushes the complexity of deciding when to prefetch a particular page to the operating system. Unfortunately, the operating system does not have enough information to make a wise decision, since it can only look at what the program has accessed in the past and has no way of finding out for sure what it is going to do in the future.

A better way to do the above is to have the compiler place the **prefetch** and **release** function calls at appropriate places in the code so that the data is present in memory at the time it is required by the application, irrespective of the dynamic behavior of the program. The only caveat is that this placement of calls by the compiler is dependent on how fast the system can bring the required page into memory, and this must be passed as a parameter to the compiler. This will potentially require each application to be recompiled for each target system according to the latency observed on it. Moreover, if this figure is very different from what it should be, the performance takes a hit. If it is higher, the compiler inserts the **prefetch** calls before time, and the prefetched pages might be swapped out before they are used. Similarly, if the time is too low, the prefetched page is not brought in by the time it is needed, and some wait time is still observed.

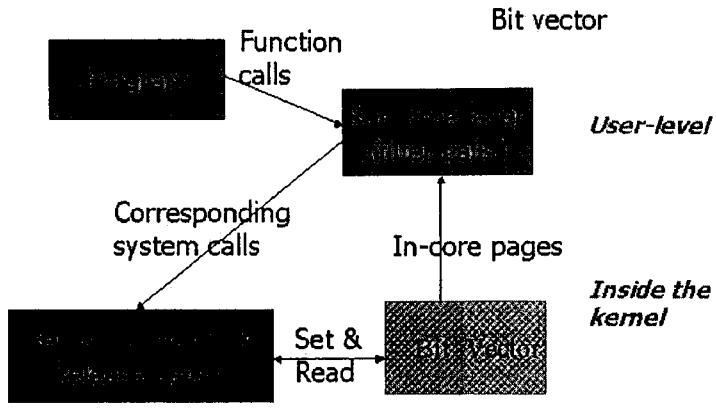


Figure 3.1: **System block diagram**

3.1.2 Operating System Support

The second thing required is operating system support for handling **prefetch** and **release** calls. One option here is to have the calls mapped to an existing interface, and not map them to a new system call. The problem with this approach is that the prefetch and release calls need to be non-binding [57] so that the prefetch requests do not compel the operating system to perform a synchronous I/O access. Thus, we would like **prefetch** and **release** to be non-binding performance hints, that are trapped and executed accordingly by an OS module. No other system call offers this exact functionality. Therefore, compiler-inserted *prefetch* and *release* calls need to be mapped to new system calls.

3.1.3 User-Level Library

The above solution does not take into account the fact that resources available at run-time may vary greatly, and that a compile-time analysis cannot completely account for the conditions faced during execution. Since it has no idea of what things will be like at run-time, the compiler is extremely aggressive in inserting prefetch calls in the code, many of which may be found unnecessary at run-time because the required pages are already present in memory. These extra calls can then cause a high overhead due to unnecessary context switches between the user and kernel space. To reduce the large context switch overhead caused by unnecessary prefetches, we need some mechanism to filter out unnecessary prefetches at the user level itself. In this architecture, this is done by a run-time layer that shares a *bit vector* with the operating system. For each page in the user's address space, the bit vector uses a bit to represent whether the page is present in memory or not. This layer filters out all unnecessary prefetch requests (for pages whose bit is already set in the bit vector) and thus manages to reduce the overhead of prefetch calls.

Therefore, in all, three main layers are required in this architecture, and the block diagram of how the instrumented program interacts with the kernel and the run-time level is shown in Figure 3.1.

3.2 Implementation Details for Linux

This section describes how we implemented the above generic, platform independent architecture on Linux. We describe the major issues we faced, and also the implementation of each of the major components.

3.2.1 Analogous System Calls

Linux-2.4.x kernel has the `madvise()` call, which offers similar functionality to the *prefetch* and *release* calls we are trying to implement. `madvise()` provides an interface by which the programmer can give hints to the operating system about what kind of future data accesses the program is to make. The relevant `madvise()` flags are *MADV_WILLNEED* and *MADV_DONTNEED*. However, the actual implementation of these calls is different from the functionality we need for our *prefetch* and *release* calls. In particular,

- MADV_DONTNEED called on a set of pages means that the pages are just zeroed, and are not written back in case they are dirty. This is something we don't want.
- MADV_WILLNEED does the prefetching from the file cache and the disk only, and does not do it for swap space.

3.2.2 Bit-vector Size

The second major issue arises from Linux's small 4K virtual memory page size (as opposed to IRIX, for example, for which it is 16K). This correlates to the size of the bit-vector that the run-time layer needs to keep track of presence/absence of virtual memory pages in the following way:

Size of user-addressable virtual memory in Linux	$= 2^{30}$ B
Virtual memory page size	$= 2^{12}$ B
No. of virtual memory pages	$= 2^{18}$
Therefore, no. of bits in bit vector	$= 2^{18}$
Virtual memory pages used by bit vector	$= 2^{3}$
Memory used by bit-vector	= 32 K

We believe a 32 K overhead is not too big, but might become a problem for multi-programmed systems or systems where available memory is low. One way to get around this is to make each bit represent 8 pages instead of just one; then, the bit can be set for example only if all 8 pages are in memory. This will result in some unnecessary prefetch calls and potentially reduce performance. This trade-off is something we haven't explored.

3.2.3 System Call Implementation

Following are very high level details of how we have implemented the system call handlers in the Linux kernel.

Prefetch Call The *prefetch* call is implemented in a fashion similar to how page faults are handled. The major difference is that these must be non-blocking. Whenever this system call is made with the number of pages that needs to be prefetched and the starting address as main parameters, the function first finds the corresponding *virtual memory area(s)* that these pages belongs to. Next, each page is handled differently depending on whether a file is associated with the page, it is an anonymous page, or whether the page is present in the swap. In case there is an associated file, some wrapper functions are called, which finally result in a call to *add_to_page_cache*, which asynchronously reads in a page from the disk and adds it to the *page cache*. If the page is present in the swap, a call to *read_swap_cache_async*, with *wait* parameter '1' is made. The wait parameter '1' ensures that the function does not block till the page is completely read from the file. In case of an anonymous page, we do nothing, because we felt that the actual latency at the time of page fault would not be very much since no data had to be read from the disk.

3.2.4 Release call

In case of a *release call*, we call the *pte_mkold* macro so that the *_PAGE_ACCESSED* bit of the page frame is set to zero. This ensures that the next time the page is scanned by *kswapd()* (a kernel daemon that scans the pages in the memory regularly and discards the infrequently used pages), its *page table entry* is dropped.

3.2.5 Instrumentation

The efficiency of our prefetching system is largely dependent on whether the prefetch requests are issued at the right time or not. To find out the actual run-time details, we keep additional information in a page. We use 3 of the unused bits in the *struct page* structure for this purpose. Specifically, we set bit 21 (*PG_prefetched*) whenever a prefetch request is made for a page. Similarly, we use bit 22 (*PG_released*) to denote the issue of release call and bit 23 (*PG_accessed*) to denote that the page has been accessed at least once since it was last brought into memory. Using these 3 bits and a set of counters that we maintain for each running process (that has registered for prefetching), we profile the page-faults and the evicted pages. There are two possibilities when a page is brought in through the page fault mechanism - either the prefetch request has been issued (*a prefetched fault*) or it has not been issued (*a non-prefetched fault*).

We maintain a counter for each of these. The run-time layer would filter the release calls for any page that is not present in the main memory, and hence any faulting page cannot have an outstanding release call. Similarly, when a page is evicted, it would have one or more of the above three bits set. Since *release calls* can only be issued after the *prefetch calls*, we have 6 possible cases and we have one counter to track each case. So, in all, we have 8 counters keeping track of all the profiling details. The *PG_accessed* bit should ideally be set whenever there is a TLB miss, but, since we could find no way of doing that, we set it whenever a page is accessed by the page fault mechanism, or, has *PG_referenced* bit set (true whenever a reference is made through the *file cache*). The *PG_prefetched* and *PG_released* bits are set whenever the corresponding system call for that page is successfully completed.

Finally, though we tested out a few cases with our instrumented kernel, we could not completely verify the functionality of the instrumentation. In particular, the setting and unsetting of *PG_accessed* bits is a little shaky, and so our results are not completely reliable.

3.2.6 Pitfalls and Workarounds

The first implementation issue we faced concerned the type of machine we should do our development and testing on. The benefits of prefetching are most obvious when the disk I/O subsystem has a high-bandwidth, since this ensures that a fast processor does not outrun all our prefetch calls. The natural solution is to use a RAID array, and we obtained access to such a machine within our research group. Also, since we could not perform large-scale reconfiguration of the machine (in particular, configuring the RAID array as swap), we used memory-mapped files as the backup store for our benchmarks. This required some extra effort on our part, as the benchmarks had to be re-written to use memory-mapped files. Also, such high-performance machines are usually heavily used, and frequent kernel recompilations and crashes are not an option. To get around this problem, we decided to perform development and initial testing using the *User Mode Linux (UML)* program, which enables the running of a full-fledged Linux kernel on a virtual machine in user space. This proved to be of limited use, however, because any benchmark that stressed UML's virtual memory system took so long to run that we could not get any useful results out of it.

Since we could not use UML, to enable quick development, we needed to run our benchmarks quickly on our experimental kernels. We also wanted our tests to stress the virtual memory sub-system. With available physical memory of 512 MB, this meant that our benchmarks should use an available data set of at least 1 GB which would translate into a running times ranging from at least a few hours to a few days depending on the benchmark. In such a situation, we found it extremely useful to *lock* the memory so that, less memory is available to the running application and we could run the tests faster.

Also, we did not port the compiler itself that inserts the prefetch and release calls, since we could get away with the compiler, implemented as a set of SUIF passes, running on IRIX and porting the resultant C code to Linux. Moreover, we could not justify devoting time out of an operating system course project to porting the compiler. Finally, we believe that it should not be difficult since the compiler passes are implemented in SUIF, which is available on a wide-variety of platforms and is largely platform-independent.

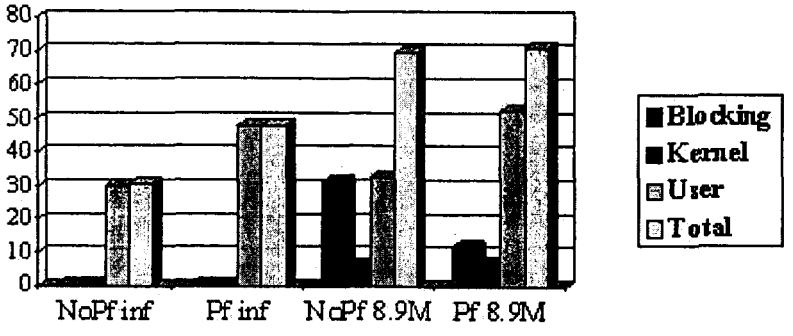
3.3 Experiments and Results

3.3.1 Hardware Set-up

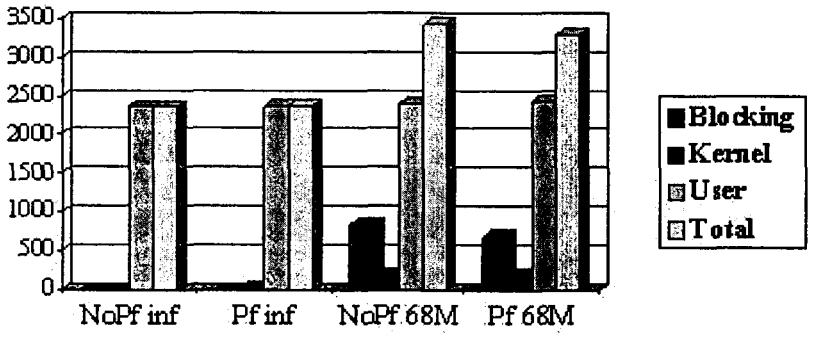
To test our implementation, we have used a CS project machine (`konzelmann.stampede.cs.cmu.edu`) with the following hardware configuration:

1. Dual Intel Pentium II 550 MHz processors (of which we use only one since the benchmarks are serial)
2. 512 MB RAM
3. 3 SCSI disks, configured as a RAID0 (striping) array.

To run our benchmarks in a reasonable amount of time, we reduce the memory available to them by running a root process that locks in a large amount of memory (as much as 480 M for one benchmark) . To get a high disk I/O bandwidth, the files that act as backups for application data are placed on the RAID array. Both these steps are taken to reduce the benchmark running time to the level where we could run a sufficient number of tests to adequately test the system; the effects of a higher disk latency on system performance are explained below. Therefore, the results we have presented here are exclusively for memory-mapped file I/O.



(a) Performance of buk (data set size 64 MB), with no memory constraint, and with memory limited to 8.9 MB



(b) Performance of embar (data size 128 MB), with no memory constraint, and with memory limited to 68 MB

Figure 3.2: **Performance of the prefetching (Pf) and non-prefetching(NoPf) versions of buk and embar, with unlimited and limited memory. The vertical axes are execution times in seconds**

3.3.2 Benchmarks

We have used benchmarks from the NAS (NASA Advanced Supercomputing) Parallel Benchmark suite to test our implementation. These benchmarks are written in Fortran, and are highly regular programs for performing simple computations on large data sets. The compilation step first converts them to C, inserts the prefetch and release calls, and compiles to machine code.

Unfortunately, of the 8 benchmarks in the suite, we could only use 2 (`buk` and `embar`). With the others, we faced the following problems:

1. There was no easy way to modify 2 benchmarks (`cgm`, and `mgrid`) to use memory mapped files for storing the large arrays.
2. For others (`applu`, `appsp`, `appbt` and `fft`), there was no straightforward way to reduce data set size and bring down the execution time. In fact, for the first three, the compilation step itself took 2 to 3 hours.

The benchmarks we did use have the following characteristics:

1. `buk` is a simple bucket-sort program that sorts a set of randomly generated integers. It's data set size and

execution time can be scaled linearly, and we ran our experiments with this benchmark using data set sizes of 1 M to 64 M, with the memory available memory constrained to 8.9 M.

2. `embar` performs a Monte Carlo simulation using an array of 2^{24} randomly generated integers. This works out to 128 M of memory usage. We ran this benchmark with the available memory constrained to 68 M.

3.3.3 Performance with and without prefetch

We ran the prefetching and non-prefetching versions of the two benchmarks, on memory constrained and unconstrained systems. The results are shown in Figure 3.2. The graphs show us that the performance of `buk` remain almost the same after prefetching is inserted in the memory-constrained case, while that of `embar` improves slightly. `Embar`'s improved performance is attributable to lesser I/O blocking time, while in the case of `buk`, better I/O blocking time and worse user-mode execution time balance each other out. Each of the contributing factors for this are explored in greater detail in our remaining experiments.

3.3.4 Scaling of performance with data size

To see how the prefetching version performed, we scaled the `buk` benchmark's data-set size from 1 MB to 64 MB, and ran both the prefetching and non-prefetching versions. The results are shown in Figure 3.3, from which the following main points can be observed:

1. Time spent blocked waiting for I/O (Figure 3.3(a)) goes down considerably in the prefetching version. The jump that occurs when the program moves from in-core to out-of-core is smaller, and the time also scales at a much smaller rate. For the largest data set, the benefit is about 65%. This is, however, a smaller benefit than the one achieved in [9], and reasons for this will be discussed later.

2. The time spent in kernel mode (Figure 3.3(a)) remains the same. This is inspite of the overhead introduced by prefetch and release system calls. The reason is that the page-faults, and hence the page-fault handling time, also go down.

3. Time spent in user-mode *increases* (Figure 3.3(b)), by as much as 60%. Some of this is attributable to the loop striping and unrolling that the compiler performs in the application code, but a bulk of it is because of the call overhead for calling functions in the run-time layer for each prefetch and release, and also because each of these function executions accesses a single bit in a 32 kB chunk of memory, with the attendant cache miss overheads. This can be brought down somewhat by optimizing the code and inlining the prefetch and release calls, which we have not been able to do due to lack of time. This overhead has been dealt with further in the next sub-section.

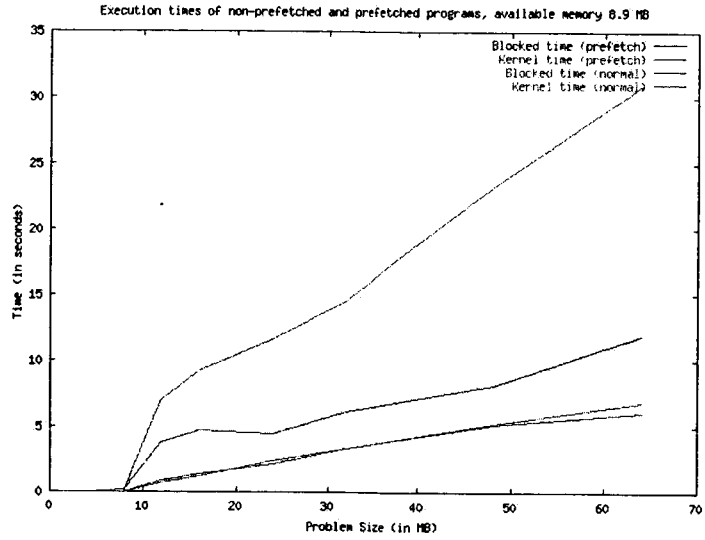
4. Finally, we find that the positive effect of (1) and the negative effect of (3) balance each other almost perfectly, and the total execution time in the memory constrained cases remains practically unchanged(Figure 3.3(b)). While this may seem discouraging, it should be noted that the overhead of the prefetch and release calls can be reduced by optimizing the implementation, and making the compiler more conservative in inserting prefetches. Further, the benefits of reduced I/O blocking time will become more apparent in more I/O intensive applications, or in those where the compiler inserts fewer prefetch instructions but which are still useful. This is true, for example, of `embar` (Figure 3.2(b)) whose performance improves.

3.3.5 Performance of the Run-time layer

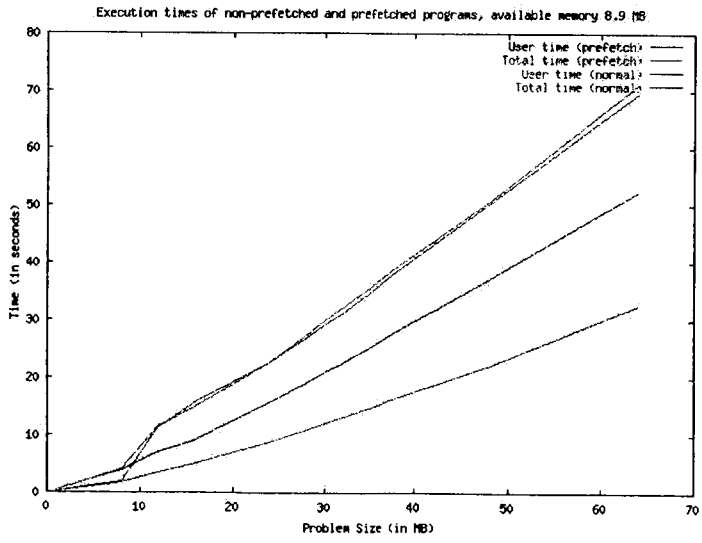
We ran two tests here: we first ran our standard prefetching version of buk, with the run-time layer filtering out unnecessary prefetch and release calls. We then ran our tests with the user-level run-time layer disabled (i.e., passing all prefetch and release calls to the kernel and not filtering any out), and measured the impact of this on observed performance.

Table 3.1 shows how many `prefetch` and `release` calls are encountered at run-time, and what the run-time layer does with them. Two cases are presented, the application running with no memory restrictions (labeled Inf) and with only 8.9 MB available to it (labeled 8.9M).

It can be seen from the table that the number of prefetch and release calls encountered scales almost linearly with the size of the benchmark's data set. For the prefetch calls, the run-time layer filters out a large majority of the calls:



(a) Time spent in kernel mode, and waiting for I/O



(b) Time spent in user mode, and total time spent

Figure 3.3: **Performance of buk for different problem sizes, with memory constrained to 8.9 MB**

Table 3.1: **Performance of the user-level run-time layer for the buk benchmark. Two cases are presented: the benchmark running with no memory restrictions (Inf) and with only 8.9M available.**

Problem Size (MB)	Prefetch Issued	Prefetch Passed (Inf)	Prefetch Passed (8.9M)	Release Issued	Release Dropped (Inf)	Release Dropped(8.9M)
1	1935890	8	14	770	30	98
8	20292967	100	136	7767	20	66
24	62252389	294	5083	24149	12	31
64	167151269	784	14001	65429	12	36

Table 3.2: **Total and kernel-mode execution time for buk, with and without run-time layer functionality. All times are in seconds (buk, 64 MB data size, 8.9 MB available memory)**

	With run-time layer		Without run-time layer	
Problem Size (MB)	Total time	Kernel time	Total time	Kernel time
1	0.39	0.00	16.26	15.28
8	4.21	0.01	169.66	160.33
24	22.54	2.43	495.01	462.18
64	71.20	6.83	1439.92	1344.94

more than 99.99% for every data size. However, the number of calls that are not dropped show an interesting trend: the number is always larger for the memory restricted case than for the memory unrestricted case, and shows a sharp jump between the 8 MB data set and the 24 MB data set: as the application moves from in-core to out-of-core, the number of pages missing from the memory, and which need to be prefetched, rises sharply. The number for the unrestricted case, on the other hand, grows almost linearly.

The release calls filtered out are more for the memory restricted case than the memory unrestricted one: a page is more likely to have been already swapped out, and therefore not requiring a release, if the available memory is limited.

These figures also help explain why the user-level running time is so much for the prefetching than for the non-prefetching case. For the largest data set (64 MB), the application makes almost 170 million function calls to routines in the run-time-library. Each of these routines has the minimal functionality of accessing a bit in a 32 kB chunk of memory. If the function call overheads, likely cache misses etc. are accounted for, the nearly 60% increase in user time can be understood.

To measure the benefit that the user-level layer gives us, we ran one benchmark (buk with a data set scaled from 1 to 64 MB, and constrained to use 8.9 MB of memory) with the run-time layer not doing any filtering, i.e., passing on all the prefetch and release calls to the kernel. The execution time for the benchmark, and time spent in kernel mode, are given in Table 3.2. As can be seen, the execution time just blows up, which is not surprising given the fact that for the largest data size, there are more than 167 million system calls taking place, each involving a context switch from the benchmark process to the kernel, and back. This overhead is tremendously reduced with the run-time layer doing it's job. Note that not filtering any prefetch and release calls does not increase actual paging activity or cause incorrect execution, since the system call handlers for prefetch and release check for the presence or absence of pages anyway. However, user-mode execution time does go up by a factor of two; this is probably because some of the context switch overhead gets accounted for as user-mode time.

3.3.6 Performance of the compiler

The compiler has to insert prefetch instructions for the correct piece of memory at the correct place in the code, so that the required data can be transferred into memory by the time it is actually needed, and not so early that it is pushed back out without being accessed at all, by a subsequent page-fault. Inserting prefetches for the correct data depends on the sophistication of the compiler's access analysis algorithms; the correct place for inserting a prefetch is determined by the system's mean page fault servicing time in terms of executable instructions.

Table 3.3: **Statistics on how well the compiler does in inserting the correct prefetch instructions (buk, 64 MB data size, 8.9 MB available memory)**

Pages accessed and prefetched	260978
Pages accessed and not prefetched	1347
Pages prefetched and accessed	257285
Pages prefetched and not accessed	3693

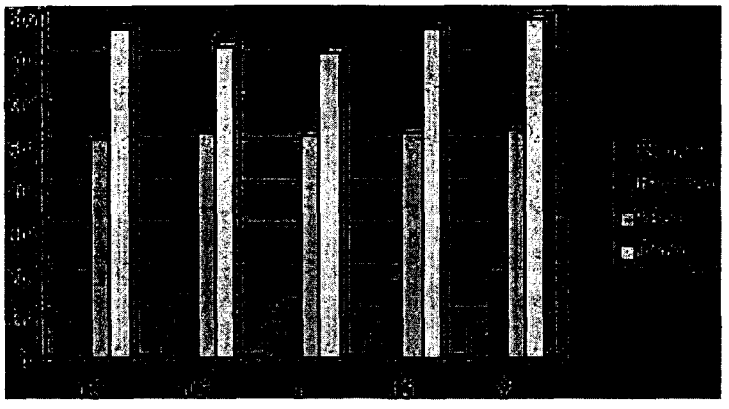


Figure 3.4: **Variation in execution time with how soon or late prefetches are inserted. 1 represents the optimum (buk, 64 MB data size, 8.9 MB available memory)**

To determine if this was indeed the case would require extensive instrumentation of the kernel, which we were not able to complete. The results of the little instrumentation we did do are shown in Table 3.3. This data is for buk with 64 MB data size and 8.9 MB available memory. The data shows that the compiler is indeed able to predict most of the data accesses, and also predicts very few wrong ones. Therefore, the I/O wait time that the benchmark still sees is because all the transfer latency cannot be hidden and some pages are brought in too late.

We also varied the parameter provided to the compiler to indicate how soon to insert prefetches, and measured the corresponding impact on execution time. This data is shown in figure 3.4. The value of 1 represents the optimum, the other values are corresponding multiples of the optimal. As can be seen, the I/O blocking time and therefore the execution time follow a U-shaped curve. Note that the optimal found by us does not correspond to the value that would mask all the I/O latency; in that case, our I/O wait time should have reduced to 0. With larger values of the parameter, the blocking time starts increasing instead of going down because our memory is too constrained, and therefore some pages start getting removed even before they are accessed. We believe a combination of a higher value for this parameter, larger available memory and/or a smaller application data set should cover practically all the I/O latency; however, we have not had time to confirm this.

3.3.7 Discussion

From these experiments, we have shown the performance of our prefetching system, and shown the components which improve in performance (I/O blocking time), and also those which lose out (user-mode execution time). Here, we show how these benefits and overheads measure up against the original implementation presented in [9].

1. **I/O wait time**: For the one benchmark that we tested extensively, buk, we reduced I/O wait time by about 60%, against a reduction of almost 100% achieved by Brwon et al. We believe our reduced reduction is largely an artifact of the extremely limited memory we use (the case we have used for our measurements has a data set size 720% of available memory, while the original implementation used a ration of only 215%). With more available memory, the prefetches could be scheduled earlier, potentially hiding much of the remaining I/O latency.

2. **User-mode execution time**: While the increase under this head for our benchmark runs seems bad, it should be noted that Brown et al also show similar overheads; they account for it differently by calling it "Prefetch Overhead".

3. **Total Execution time**: Our test run for `buk` achieved no speed-up, while Brown et al. report a speed-up of 68%. This is because I/O wait time forms a very large component of their execution time, and whatever reduction is achieved there reflects in the total execution time. In our case, I/O latency is so small to start with (see fig 3.2(a)) that the 65% reduction achieved is only enough to cover increase in user-mode execution time.

We would also like to point out that `buk` is probably an extreme case since the compiler inserts so many prefetches; our results for `embar` show a greater similarity to the ones in [9].

3.4 Conclusions Regarding Compiler-Based Prefetching in Linux

We have shown that implementation of a prefetching system on Linux is feasible, and is capable of giving reasonably good performance improvements. We have performed a number of experiments and tests to show the effectiveness of each component of the architecture, and in particular that of the run-time layer. We have also performed a detailed analysis of where our system meets our expectations, where it does not, and why it fails when it does.

We would also like to take this opportunity to enumerate some of the lessons we have learned from this effort:

1. Functioning systems are complex entities, and trying to understand such a system and improve it is a hard task. In particular, there is always a hidden dependency somewhere that had not been budgeted for.

2. A potential for improvement in performance does not always translate to actual benefits achieved, especially if the implementation is inefficient.

3. Testing and performance evaluation are often the most time-consuming components of system-building, and any work plan should always allocate enough time to these.

PART II: Thread Level Data Speculation

Chapter 4

Thread-Level Speculation Background

4.1 Introduction

Due to rapidly increasing transistor budgets, today's microprocessor architect is faced with the pleasant challenge of deciding how to translate these extra resources into improved performance. In the last decade, microprocessor performance has improved steadily through the exploitation of *instruction-level parallelism* (ILP), resulting in superscalar processors that are increasingly wider-issue, out-of-order, and speculative. However, this highly-interconnected and complex approach to microarchitecture is running out of steam. Cross-chip wire latency (when measured in processor cycles) is increasing rapidly, making large and highly interconnected designs infeasible [61, 1]. Both development costs and the size of design teams are growing quickly and reaching their limits. Increasing the amount of on-chip cache eventually shows diminishing returns [47]. Instead, an attractive option is to exploit *thread-level parallelism* (TLP).

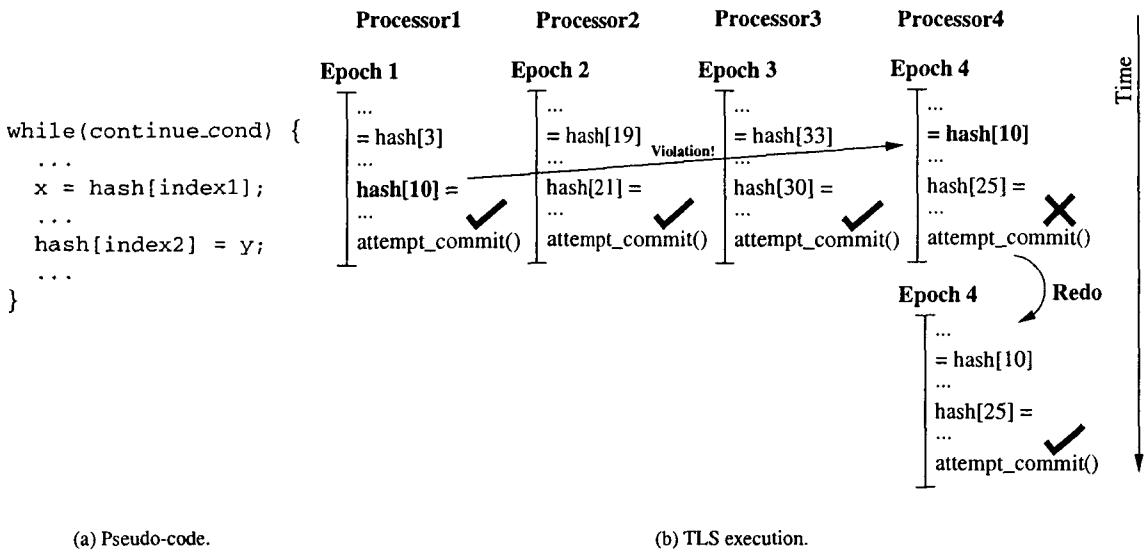
The transition to new designs that support multithreading has already begun: the Sun MAJC [72], IBM Power4 [42], and the Sibyte SB-1250 [22] are all *chip-multiprocessors* (CMPs), in that they incorporate multiple processors on a single die. Alternatively, the Alpha 21464 [29] supports *simultaneous multithreading* (SMT) [74, 75, 28], where instructions from multiple independent threads are simultaneously issued to a single processor pipeline. These new architectures still benefit from ILP, since ILP and TLP are complementary.

While it is relatively well-understood how to design cost-effective CMP and SMT architectures, the real issue is how to use thread-level parallelism to improve the performance of the software that we care about. Multiprogramming workloads (running several independent programs at the same time) and multithreaded programs (using separate threads for programming convenience, such as in a web server) both naturally take advantage of the available concurrent threads. However, we often are concerned with the performance of a single application. To use a multithreaded processor to improve the performance of a single application we need that application to be *parallel*.

Writing a parallel program is not an easy task, requiring careful management of communication and synchronization, while at the same time avoiding load-imbalance. We would instead like the compiler to translate any program into a parallel program automatically. While there has been much research in this area of automatic parallelization for *numeric* programs (array-based codes with regular access patterns), compiler technology has made little progress towards the automatic parallelization of *non-numeric* applications: progress here is impeded by ambiguous memory references and pointers, as well as complex control and data structures—all of which force the compiler to be conservative. Rather than requiring the compiler to prove independence of potentially-parallel threads, we would like the compiler to be able to parallelize if it is *likely* that the potential threads are independent. This new form of parallel execution is called *thread-level speculation* (TLS).

4.2 Example

TLS allows the compiler to automatically parallelize portions of code in the presence of statically ambiguous data and control dependences, thus extracting parallelism between whatever dynamic dependences actually exist at runtime. To illustrate how TLS works, consider the simple `while` loop in Figure 4.1(a) which accesses elements in a hash table. This loop cannot be statically parallelized due to possible data dependences through the array `hash`. While it is possible that a given iteration will depend on data produced by an immediately preceding iteration, these



(a) Pseudo-code. (b) TLS execution.

Figure 4.1: Example of thread-level speculation (TLS).

dependences may in fact be infrequent if the hashing function is effective. Hence a mechanism that could speculatively execute the loop iterations in parallel—while squashing and re-executing any iterations which do suffer dependence violations—could potentially speed up this loop significantly, as illustrated in Figure 4.1(b). In this example, the program is running on a shared-memory multiprocessor, and some number of processors (four, in this case) have been allocated to the program by the operating system. Each of these processors is assigned a unit of work, or *epoch*, which in this case is a single loop iteration. When complete, each epoch attempts to commit its speculative work. In this case a *read-after-write* (RAW) data dependence violation is detected between *epoch 1* and *epoch 4*; hence *epoch 4* is squashed and restarted to produce the correct result, while *epochs* 1, 2, and 3 commit. This example demonstrates the basic principles of TLS.

4.3 Execution Model

This section describes the execution model for TLS that is targeted by the compiler and implemented in hardware. The following description also serves as a high-level overview of the the remainder of this section.

First, we divide a program into speculatively-parallel units of work called *epochs*. Each epoch is associated with its own underlying speculative thread, and creates the next epoch through a lightweight fork called a `spawn`, as shown in Figure 4.3(a). The spawn mechanism forwards initial parameters and a program counter (PC) to the appropriate processor. An alternative approach would be to have a fixed pool of speculative threads that grab epochs from a centralized work queue.

A key component of any architecture for TLS is a mechanism for tracking the relative ordering of the epochs. In our approach, we timestamp each epoch with an *epoch number* to indicate its ordering within the original sequential execution of the program. We say that *epoch X* is *"logically-earlier"* than *epoch Y* if their epoch numbers indicate that *epoch X* should have preceded *epoch Y* in the original sequential execution. Epochs commit speculative results in the original sequential order by passing a *homefree token* which indicates that all previous speculative threads have made all of their speculative modifications visible to the memory system and hence it is safe to commit. When an epoch is guaranteed not to have violated any data dependences with logically-earlier epochs and can therefore commit all of its speculative modifications, we say that the epoch is *homefree*.

In the case when speculation fails for a given epoch, all logically-later epochs that are currently running are also violated and squashed. In Figure 4.3(b), speculation fails for epoch 2 which in turn causes epoch 3 to be squashed. Although more aggressive strategies are possible, this conservative approach ensures that an epoch does not continue to execute when it may have received incorrect data.

Epoch: The unit of execution within a program which is executed speculatively.

Epoch Number: A number which identifies the relative ordering of epochs within an OS-level thread. Epoch numbers also indicate that certain parallel threads are unordered.

Homefree Token: A token which indicates that all previous epochs have made their speculative modifications visible to memory and hence speculation for the current epoch is successful.

Logically Earlier/Later: With respect to epochs, *logically-earlier* refers to an epoch that preceded the current epoch in the original execution while *logically-later* refers to an epoch that followed the current epoch.

OS-level Thread: A thread of execution as viewed by the operating system—multiple speculative threads may exist within an OS-level thread.

Speculative Context: The state information associated with the execution of an epoch.

Sequential Portion: The portion of a program where TLS is not exploited.

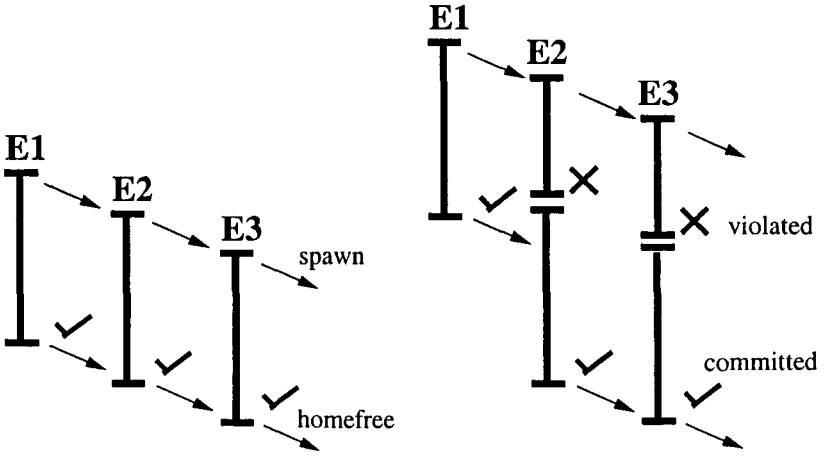
Spawn: A light-weight fork operation that creates and initializes a new speculative thread.

Speculative Region: A single portion of a program where TLS (speculative parallelism) is exploited.

Speculative Thread: A light-weight thread that is used to exploit speculative parallelism within an OS-level thread.

Violation: A thread has suffered a true data dependence violation if it has read a memory location that was later modified by a logically-earlier epoch—other types of violations are described later.

Figure 4.2: Glossary of terms.

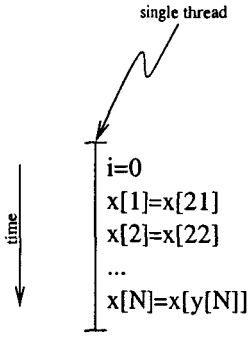


(a) Steady-state execution.

(b) Recovering from a violation.

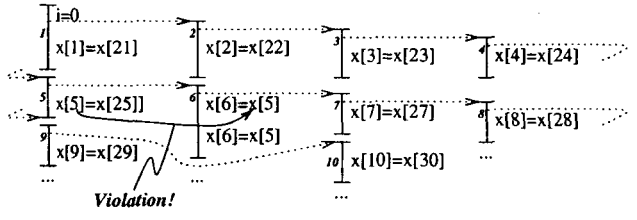
Figure 4.3: TLS execution model.

```
/* Initialize y so that there is a RAW dependence
 * in iteration 6 of the loop: */
y[] = {20, 21, 22, 23, 24, 25, 5,
     27, 28, 29, 30, ...};
...
i = 0;
while (++i < N) {
  x[i] = x[y[i]];
}
```



(a) The sequential version of a simple `while` loop.

```
  my_i = 0;
start(my_i):
  if(++my_i < N) {
    /* Pass value of my_i as a parameter to the
     * next thread: */
    td = lightweight_fork(&start, my_i);
    x[my_i] = x[y[my_i]];
    attempt_commit();
    end_thread();
  }
  /* Falls through after last iteration: */
  i = my_i;
```



(b) The speculative version.

Figure 4.4: Speculative parallelization of a simple loop.

4.4 Software Interface

We have architected our TLS system to involve both the compiler and hardware, hence we require an interface between them. There are a number of issues to consider for such an interface: some issues are analogous to those for purely parallel applications, such as creating threads and managing the stacks; others are unique to TLS, such as passing the *homefree* token and recovering from failed speculation. In this section, we begin with a description of the important components of the software interface to TLS hardware, and then we present the new instructions that implement this interface.

4.4.1 Required Interface Mechanisms

For thread-level speculation, there are many possible implementations for the interface between hardware and software. In this section we briefly explore the design space of interfaces and also present our approach.

Threads

Before we discuss issues which are unique to speculation, we first consider one of the requirements for any kind of parallel execution: the creation of parallel threads. Figure 4.4(a) shows an example of a simple `while` loop where the elements of the array `x` are updated using array `y` as an index into `x`. The array `y` has been initialized such that the sixth iteration of the loop (i.e. when `i` = 6) will depend on the output of the preceding iteration (i.e. when `i` = 5); otherwise, the iterations are independent.

There are two methods for distributing work to threads: a static method, and a dynamic method. With static distribution, all threads are initialized prior to speculative execution—for example, by specifying a round-robin assignment of epochs for the next speculative region. With dynamic creation, each speculative thread initializes the next; this

```
i = 0;
do {
   x[i] = x[y[i]];
   i++;
} while (i < x[i]);
```

(a) A simple do-while loop with unknown bounds.

```
   my_i = 0;
start(my_i):
   td = lightweight_fork(&start, my_i + 1);
   x[my_i] = x[y[my_i]];
   if(cancelled()) {
      /* Cascade the cancel to subsequent threads: */
      cancel_thread(td);
      end_thread();
   }
   attempt_commit();
   my_i++;
   if(my_i < x[my_i]) {
      end_thread();
   }
   /* Cancel extra loop iterations and continue: */
   cancel_thread(td);
   i = my_i;
```

(b) Speculative version with cancelling.

Figure 4.5: Interface for cancelling superfluous epochs.

approach is more tolerant of load imbalance between epochs than the static approach. The dynamic approach is also advantageous when the number of available processors is constantly changing, (e.g. in a multiprogramming environment): the dynamic approach can more easily adapt to this situation by growing or shrinking the number of threads to match the available resources.

Figure 4.4(b) illustrates the dynamic model through a simple loop. Each epoch creates the next through a lightweight fork operation, performs its speculative work, and then attempts to commit its speculative modifications to memory. The lightweight fork returns a thread descriptor (`td`) which serves as a handle on the next thread.

Stacks

A key design issue is the management of references to the stack. A naive implementation would maintain a single stack pointer (shared among all epochs) and a stack in memory that is kept consistent by the underlying data dependence tracking hardware. The problem with this approach is that speculation would fail frequently and unnecessarily: for example, whenever multiple epochs store values to the same location on the stack. These dependence violations would effectively serialize execution. In addition, whenever the stack pointer is modified, the new value must be forwarded to all successive epochs. An alternative approach is to create a separate *stacklet* [36] for each epoch to hold local variables, spilled register values, return addresses, and other procedure linkage information. These stacklets are created at the beginning of the program, assigned to each of the participating processors, and re-used by the dynamic threads. The stacklet approach allows each epoch to perform stack operations independently, allowing speculation to proceed unhindered.

Epoch Cancellation

To speculatively parallelize certain code structures we require support for *control speculation*. For example, a `while` loop with an unknown termination condition can be speculatively parallelized, but superfluous epochs beyond the correct termination of the loop may be created. For correct execution we require the ability to cancel any such superfluous epochs. Figure 4.5(a) shows a `while` loop with an unknown ending condition, and Figure 4.5(b) shows the speculatively-parallelized version of the loop with support for cancelling superfluous epochs. Each speculative context provides a flag that indicates whether the current epoch has been cancelled. Any epoch that is cancelled also cancels its child epoch if one exists.[1] In contrast to an epoch that suffers a violation, a cancelled epoch is not re-executed by the run-time system.

[1] Another option is to interrupt and terminate the epoch, rather than poll a flag.

```
i = 0;
while (++i < N) {
  x[i] = x[y[i]];
}
```

(a) Original code.

```
  my_i = 0;
start(my_i):
  if(++my_i < N) {
    td = lightweight_fork(&start, my_i);
    x[my_i] = x[y[my_i]];
    wait_for_homefree_token();
    commit_speculative_writes();
    pass_homefree_token(td);
    end_thread();
  }
  /* Falls through after last iteration: */
  i = my_i;
```

(b) Threaded code.

Figure 4.6: Interface for passing the homefree token.

Homefree Token

We cannot determine whether speculation has succeeded for the current epoch until all previous epochs have made their speculative modifications visible to memory—hence the act of committing speculative modifications to memory must be serialized. Two unattractive options would be 1) for a central entity to maintain the ordering of the active epochs, or 2) to broadcast to all processors any changes in the allocation of epoch numbers. A more efficient approach is to directly pass an explicit token—which we call the *homefree token*—from the logically-earliest epoch to its successor when it commits and makes all of its speculative modifications visible to memory. Receipt of the homefree token indicates that the current epoch has no predecessors, and hence is no longer speculative.

Figure 4.6 illustrates how the `attempt_commit` operation can be implemented by waiting for the homefree token, committing speculative modifications to memory, and then passing the homefree token directly to the next epoch—thereby avoiding the need for central coordination. This homefree token mechanism is simply a form of producer/consumer synchronization and hence can be implemented using normal synchronization primitives. One option is to perform homefree synchronization at the user level through regular shared memory, without differentiating the homefree token from other synchronization objects. Another option is to make the homefree token visible to hardware so that hardware can react immediately to its arrival. For now, our software interface defers this decision and assumes that the underlying run-time system provides the necessary synchronized behavior.

Value Forwarding

There are often variables that are, at compile time, provably dependent between epochs: for example, a local scalar that is both used and defined every epoch. There are two options when such cases arise. First, the compiler could allocate the variable in memory which would then cause dependence violations between all consecutive epochs. Second, the compiler could synchronize and forward that variable between consecutive epochs, avoiding dependence violations. In our approach, the compiler allocates forwarded variables on a special portion of the stack called the *forwarding frame*; the forwarding frame supports the communication of values between epochs, and synchronizes the accesses to these variables. The forwarding frame is defined by a base-address within the stack frame and an offset from that base address; this way, any regular load or store to an address within the predefined forwarding frame address range can be treated appropriately. The address range of the forwarding frame is defined through the software interface at the beginning of every speculative region. Accesses to the forwarding frame are exempt from the data dependence tracking mechanisms of the underlying hardware.

4.4.2 TLS Instructions

We now describe the software interface to TLS hardware support. The goal of this design is to provide the necessary interface to hardware while permitting the exploration of implementation alternatives. For this reason we decompose all TLS events to their component parts and assign a new TLS instruction to each part.

`thread_descriptor spawn(start_address)`: Creates a new thread which begins execution at the start address given. A thread descriptor is returned which can be used to refer to the thread in future calls—if no processor is able to enqueue this request for a new thread then a thread descriptor of zero is returned. A speculative context including a *stacklet* is also allocated for the new thread, and the contents of the forwarding frame are copied to the new thread's forwarding frame.

`void end_thread()`: Terminates execution of the current thread, frees its resources, frees its stack (unless it was saved), and invalidates any uncommitted speculative modifications.

`error_code cancel_thread(thread_descriptor)`: Locates the specified thread and delivers a cancel signal to it. If the thread has registered a cancel handler, the thread asynchronously jumps to it. If not, then the thread is terminated. If the specified thread does not exist then this instruction is treated as a no-op.

`error_code violate_thread(thread_descriptor)`: Notifies the specified thread of a violation—if the thread descriptor is null or invalid then this is a no-op.

`sp save_sp()`: Returns the current stack pointer and marks it so that the stacklet is not freed when the thread ends.

`void restore_sp(sp)`: Frees the current stacklet and sets the stack pointer to the value provided.

Figure 4.7: TLS instructions for thread and stack management.

`void set_sequence_number(sequence_number)`: Sets the sequence number of the current thread to create a partial ordering in relation to other speculative threads.

`void become_speculative()`: Informs the processor that subsequent memory references should be treated as speculative; if the homefree token has already arrived then this is a no-op. If a violation occurs during speculation then the processor discards all speculative modifications, returns to the address of this `become_speculative()` instruction, and restarts execution.

`void wait_for_homefree_token()`: Blocks a thread until it receives the homefree token. If the homefree token is already held then this is treated as a no-op.

`void commit_speculative_writes()`: This blocking instruction makes buffered speculative modifications visible to all other speculative threads before returning.

`void pass_homefree_token(thread_descriptor)`: This instruction passes the homefree token to the specified thread.

Figure 4.8: TLS instructions for speculation management.

`void set_forwarding_frame(forward_struct_address)`: Sets the base address of the forwarding frame (within the stack frame) When used in combination with set_forwarding_size, this specifies a portion of the stack to be copied to the child epoch upon a `spawn`, as well as individual locations to be forwarded mid-epoch.

`void set_forwarding_size(size)`: Specifies the size of the forwarding frame (as measured from the base address provided by the set_forwarding_frame instruction).

`void wait_for_value(offset)`: Causes the current thread to block until it receives a value at the specified offset in the forwarding frame.

`void send_value(thread_descriptor,offset)`: Send the scalar value from the specified location within the forwarding frame to the specified thread. The receiving thread will wake up if it is waiting for that value, and will not block should it subsequently attempt to wait for that value.

Figure 4.9: TLS instructions for value forwarding.

Instructions for Managing Threads and Stacks

A TLS program requires the ability to create speculative threads. In our approach, we are primarily concerned with providing concurrency at a very low cost—hence we implement a lightweight fork instruction called a `spawn`. A `spawn` instruction creates a new thread which begins execution at the start address (PC) given, and is initialized through a copy of the current thread's forwarding frame; a thread descriptor for the child thread is returned as a handle. When its speculative work is complete, a thread is terminated by the `end_thread` instruction. Rather than require software to be aware of the number of available speculative contexts (eg., processors), the semantics of the `spawn` primitive are such that it may fail: failure is indicated by a returned thread-descriptor value of zero. Whenever a `spawn` fails, the speculative thread that attempted the `spawn` simply executes the next epoch itself. This method allows speculative threads to grow to consume all of the available processing resources without creating an unmanageable explosion of threads.

The `cancel_thread` and `violate_thread` primitives allow an epoch to cancel another epoch or trigger a violation and recovery for another epoch, facilitating speculation on more than just data values, such as control speculation. There are also instructions for saving and restoring the stack pointer, and returning to the regular stack after using stacklets during the speculative region.

Instructions for Managing Speculation

Within a speculative thread, software must first initialize speculative execution. The `set_sequence_number` instruction allows software to specify an epoch number to hardware. After an epoch is created, it may perform non-speculative memory accesses to initialize itself. Once this phase of execution is complete, the `become_speculative` instruction indicates that future memory references are speculative.

Several instructions manage the homefree token. It is created and passed by the `wait_for_homefree_token` and `pass_homefree_token` primitives, while the `commit_speculative_writes` primitive instructs hardware to make all speculative modifications visible to the memory system before returning. All three of these instructions could potentially be combined into one instruction, but for now we keep them separate to maximize flexibility.

Instructions for Forwarding Values

There are four primitives that support the forwarding of values between epochs. First are two instructions that are executed at the beginning of a speculative region to define the forwarding frame by giving its base address and size. The other two instructions name a value to wait for or send by specifying an offset into the forwarding frame. The `send_value` primitive specifies the thread descriptor of the target epoch, and the location of the actual value to be sent. These primitives implement *fine-grained* synchronization, since we synchronize on each individual value (rather than waiting before the first use of any forwarded value and sending after the last definition of any forwarded value). This granularity also allows the processor to issue instructions out-of-order with respect to a blocked `wait_for_value` instruction. In particular, the `wait_for_value` instruction is also associated with a destination register so that the corresponding load instruction (which actually loads the value from the forwarding frame) can be blocked while other instructions are issued around it.

4.4.3 Examples

The instructions presented so far are sufficient to generate a broad class of TLS programs. Figure 4.10 shows an example loop that has been speculatively parallelized using the new TLS instructions, and the illustration shows its execution on a four processor multiprocessor. The variable `i` is updated and copied to the next epoch through the forwarding frame at each `spawn`, and also serves as the epoch number which is set by the `set_sequence_number` primitive. The code is constructed such that the `spawn` instruction may fail and the current speculative thread will continue and execute the next epoch itself. The final speculative thread will branch around the outermost `if` construct, wait to be homefree, and then continue to execute the code following the loop.

As described in Section 4.4.1, the bounds of a loop may not be known at compile time, and hence a program may erroneously continue to speculate beyond the end of the loop. In this case the extra epochs are cancelled using the `cancel_thread` instruction once the end of the loop is known. Figure 4.11 shows how such a loop with unknown bounds can be speculatively parallelized. In the example, the next iteration of the loop is always spawned, and extra iterations are cancelled when the end of the loop is found.

```
for(i = 0; i < N; i++) {
  S1;
  x[i] = y[z[i]];
  S2;
}
```

(a) A `for` loop with a possible dependence.

```
struct forwarding_frame {int i;} ff;
int my_i = 0;
set_forwarding_frame(&ff);
set_forwarding_size(sizeof(struct forwarding_frame));
ff.i = 0;
start:
my_i = ff.i;
if(my_i < N) {
  ff.i = my_i + 1;
  td = spawn(&start);
  set_sequence_number(my_i);
  become_speculative();
  S1;
  x[my_i] = y[z[my_i]];
  S2;
  wait_for_homefree_token();
  commit_speculative_writes();
  pass_homefree_token(td);
  end_thread();
}
```

(b) Transformed TLS code.

Figure 4.10: Threads in a basic for loop.

Figure 4.12 shows how we can cleverly code the speculative version of a loop so that an existing thread is re-used whenever a `spawn` fails. If the `spawn` returns zero, then the `spawn` failed and the current thread goes on to execute the next epoch itself.

Figure 4.13(a) illustrates two cross-epoch dependences: one that is ambiguous (through `x`) and one that is definite (through `v`). Our TLS interface allows us to speculate on the ambiguous dependence while directly satisfying the definite dependence through value forwarding. As shown in Figure 4.13(b), the array `x` is modified speculatively while `v` is synchronized using the TLS instructions for forwarding values. Since both `i` and `v` are stored on the forwarding frame, they have offsets of zero and one respectively. The entire forwarding frame is copied when each epoch is spawned, initializing each epoch with the proper value of `i`.

After speculatively updating `x`, each epoch must synchronize and update `v`. The `wait_for_value` primitive stalls the execution of all loads from the forwarding frame at the offset specified, and the pipeline logic also stalls any further indirectly-dependent instructions. Once the value of `v` is produced by the previous thread, the `wait_for_value` instruction unblocks and the value of `v` is loaded from the forwarding frame. The variable `v` is then updated, stored back to the forwarding frame, and then the next epoch is sent the updated value. The `send_value` instruction must not be issued out-of-order with respect to the updated definition of the variable `v`, hence it references `v` as a parameter to create a dependence.

4.5 Compiler Support

In contrast with hardware-only approaches to TLS, we rely on the compiler to define where and how to speculate. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [71] which operates on `C` code. For `Fortran` applications, the source files are first converted to `C` using `sf2c`, and then converted to SUIF format. Our infrastructure performs the following phases when compiling an application to exploit TLS.

Deciding Where to Speculate

One of the most important tasks in a thread-speculative system is deciding which portions of code to speculatively parallelize (as will be explored in Chapter 8). For the evaluations in this dissertation, the compiler uses profile information to decide which loops in a program to speculatively parallelize—a thorough treatment of this issue is beyond the scope of this dissertation. We limit our focus to loops for two reasons: first, loops comprise a significant portion of execution

```
for(i = 0; i < f(i); i++) {
  S1;
  x[i] = x[y[i]];
  S2;
}
S3;
```

(a) A `for` loop with an unknown ending condition.

```
  struct forwarding_frame {int i;} ff;
  int my_i = 0;
  if(my_i < f(my_i)) {
    set_forwarding_frame(&ff);
    set_forwarding_size(sizeof(struct forwarding_frame));
    ff.i = 0;
start:
    my_i = ff.i;
    ff.i = my_i + 1;
    set_cancel_handler(&cancelled);
    td = spawn(&start);
    set_sequence_number(my_i);
    become_speculative();
    S1;
    x[my_i] = x[y[my_i]];
    S2;
    my_i++;
    loop_test = my_i < f(my_i);
    wait_for_homefree_token();
    commit_speculative_writes();
    if (loop_test) {
      pass_homefree_token(td);
      end_thread();
    } else {
      /* This was the last iteration. */
      if(td != 0) {
        /* Cancel subsequent threads: */
        cancel_epoch(td);
      }
      goto continue;
    }
cancelled:
    /* Cancel any more epochs if they exist
    cancel_thread(td);
    end_thread();
  }
continue:
  S3;
```

(b) Transformed for TLS.

Figure 4.11: Threads in a `for` loop with an unknown ending condition.

time (coverage) and hence can impact overall program performance; second, loops are fairly regular and predictable, hence it is straightforward to transform loop iterations into epochs. Investigation of the impact of parallelizing regions other than loops is also beyond the scope of this dissertation.

The following gives a basic description of the loop selection process used to compile benchmark applications. The first step is to measure every loop in every benchmark application by instrumenting the start and end of each potential speculative region (loop) and epoch (iteration). Second, we filter the loops to only consider those that meet the following criteria:

- the coverage (fraction of dynamic execution) is more than 0.1% of execution time;
- there is more than one iteration per invocation (on average);
- the number of instructions per iteration is less than 16000 (on average);
- the total number of instructions per loop invocation is greater than 30 (on average);
- it does not contain a call to `alloca()`, which would interfere with stack management.

```
for(i = 0; i < N; i++) {
  S1;
  x[i] = y[z[i]];
  S2;
}
```

(a) A `for` loop with a possible dependence.

```
struct forwarding_frame {int i;} ff;
int my_i = 0;
if(my_i < f(my_i)) {
  set_forwarding_frame(&ff);
  set_forwarding_size(sizeof(struct forwarding_frame));
  ff.i = 0;
start:
  my_i = ff.i;
  if (my_i < N) {
    ff.i = my_i + 1;
    td = spawn(&start);
    set_sequence_number(my_i);
    become_speculative();
    S1;
    x[my_i] = y[z[my_i]];
    S2;
    wait_for_homefree_token();
    commit_speculative_writes();
    if(td == 0) {
      /* Spawn failed, re-use the thread: */
      goto start;
    } else {
      pass_homefree_token(td);
      end_thread();
    }
  }
  wait_for_homefree_token();
```

(b) Transformed TLS code.

Figure 4.12: A loop that re-uses threads.

The purpose of this initial filtering is to remove from consideration those loops that are unlikely to contribute to improved performance.

In the third step, we unroll each loop by factors of 1 (no unrolling), 2, 4, and 8, generating several versions of each benchmark to measure. Next we measure the expected performance of each loop and unrolling when run speculatively in parallel using detailed simulation on our baseline hardware support for TLS, and select loops in two different ways for the purposes of evaluating our hardware support. The first way is to maximize performance, where we select the loops that contribute the greatest performance gain—we will refer to this as the *select* version of the benchmarks. The second way is to maximize coverage, where we greedily select loops that represent the largest fraction of execution time, regardless of performance—we will refer to this as the *max-coverage* version of the benchmarks. In both cases the best performing unrolling factor is used for each loop, and chosen independently for the sequential and speculative versions of each application.

Transforming to Exploit TLS

Once speculative regions are chosen, the compiler inserts the TLS instructions (described in Section 4.4.2) that interact with hardware to create and manage the speculative threads and forward values.

Optimization

Without optimization, execution can be unnecessarily serialized by synchronization (through `wait` and `signal` operations). A pathological case is a "`for`" loop in the `C` language where the loop counter is used at the beginning of the loop and then incremented at the end of the loop—if the loop counter is synchronized and forwarded then the loop will be serialized. However, scheduling can be used to move the `wait` and `signal` closer to each other, thereby reducing this critical path. Our compiler schedules these critical paths by first identifying the computation chain leading to each `signal`, and then using a dataflow analysis which extends the algorithm developed by Knoop [43] to

```
for(i = 0; i < N; i++) {
  S1;
  x[i] = y[z[i]];
  S2;
  v += z[i];
  S3;
}
```

(a) A for loop with both definite and ambiguous dependences.

```
struct forwarding_frame {
  int i; /* offset 0 */
  int v; /* offset sizeof(int) */
} ff;
int my_i = 0;
int my_v = 0;
set_forwarding_frame(&ff);
set_forwarding_size(sizeof(struct forwarding_frame));
ff.i = 0;
ff.v = 0;
start:
my_i = ff.i;
if (my_i < N) {
  ff.i = my_i + 1;
  td = spawn(&start);
  set_sequence_number(my_i);
  become_speculative();
  S1;
  x[my_i] = y[z[my_i]];
  S2;
  wait_for_value(sizeof(int));
  my_v = ff.v;
  my_v += z[i];
  ff.v = my_v;
  send_value(td,sizeof(int));
  S3;
  wait_for_homefree_token();
  commit_speculative_writes();
  pass_homefree_token(td);
  end_thread();
}
wait_for_homefree_token();
```

(b) Transformed TLS code.

Figure 4.13: A loop requiring forwarding.

schedule that code in the earliest safe location. We can do even better for any loop induction variable that is a linear function of the loop index; the scheduler hoists the associated code to the top of the epoch and computes that value locally from the loop index, avoiding any extra synchronization altogether. These optimizations have a large impact on performance [83], as we show later in Chapters 6.

Code Generation

Our compiler outputs C source code which encodes our new TLS instructions as in-line MIPS assembly code using gcc's "asm" statements. This source code is then compiled with gcc v2.95.2 using the "-O3" flag to produce optimized, fully-functional MIPS binaries containing new TLS instructions.

4.6 Overview

In the following four chapters we explore four methods for optimizing the performance of thread-level speculation. The first three chapters investigate methods for improving the communication of values between speculative threads: in Chapter 5 we consider hardware techniques, in Chapter 6 we focus on compiler techniques for scalar values, and in Chapter 7 we focus on compiler techniques for memory-resident values.

Chapter 5

Hardware Support for Improving Value Communication

5.1 Introduction

In the context of TLS, value communication refers to the satisfaction of any true (read-after-write) dependence between *epochs* (sequential chunks of work performed speculatively in parallel). From the compiler's perspective, there are two ways to communicate the value of a given variable. First, the compiler may speculate that the variable is not modified (Figure 5.1(a)). However, if at run-time the variable actually *is* modified then the underlying hardware ensures that the misspeculated epoch is re-executed with the proper value. This method only works well when the variable is modified infrequently, since the cost of misspeculation is high. Second, if the variable is frequently modified, then the compiler may instead synchronize and forward[1] the value between epochs (Figure 5.1(b)). Since a parallelized region of code will contain many variables, the compiler will employ a combination of speculation and synchronization as appropriate. [2]

To further improve upon static compile-time choices between speculating or synchronizing for specific memory accesses, we can exploit dynamic run-time behavior to make value communication more efficient. For example, we might exploit a form of *value prediction* [3, 34, 45, 49, 63, 64, 78], as illustrated in Figure 5.1(c). To get a sense of the potential upside of enhancing value communication under TLS, let us briefly consider the ideal case. From a performance perspective, the ideal case would correspond to a value predictor that could perfectly predict the values of any inter-thread dependences. In such a case, speculation would never fail and synchronization would never stall. While this perfect-prediction scenario is unrealistic, it does allow us to quantify the potential impact of improving value communication in TLS. Figure 5.2 shows the impact of perfect prediction on several speculatively-parallelized SPECint [23] benchmarks, running on a 4-processor CMP that implements our TLS scheme [67] (details are given later in Section 5.3.2). Each bar shows the total execution time of all speculatively-parallelized regions of code, normalized to that of the corresponding original sequential versions of these same codes. As we see in Figure 5.2, efficient value communication often makes the difference between speeding up and slowing down relative to the original sequential code. Hence this is clearly an important area for applying compiler and hardware optimizations.

5.1.1 Techniques for Improving Value Communication

Given the importance of efficient value communication in TLS, what solutions can we implement to approach the ideal results of Figure 5.2? Figure 5.1 shows the spectrum of possibilities: i.e. speculate, synchronize, or predict. In our baseline scheme, the compiler synchronizes dependences that it expects to occur frequently (by explicitly "forwarding" their values between successive epochs), and speculates on everything else. How can we use hardware to improve on this approach? Hardware support for efficient *speculation* has already been addressed in a number of papers on TLS [3, 19, 37, 39, 40, 44, 48, 59, 67, 73]. Therefore our focus in this chapter is how to exploit and enhance the remaining spectrum of possibilities (i.e. *prediction* and *synchronization*) such that they are complementary to speculation within TLS. In particular, we explore the following techniques:

[1] This is also known as *doacross* [26, 60] parallelization.

[2] Results shown in this section can also be found in our previous publication. [68]

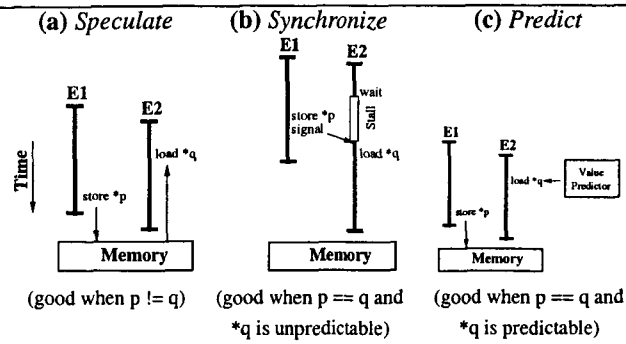


Figure 5.1: A memory value may be communicated between two epochs (E1 and E2) through (a) speculation, (b) synchronization, or (c) prediction.

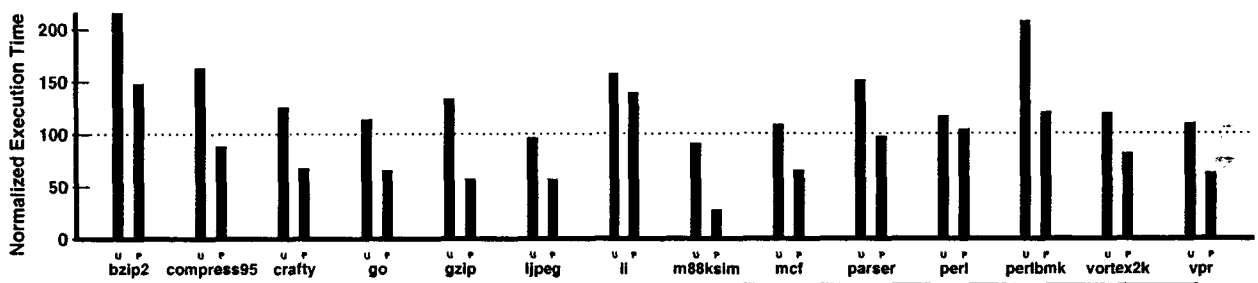


Figure 5.2: Potential impact of optimizing value communication. Relative to the normalized, original sequential version, *U* shows the unoptimized speculative version and *P* shows perfect prediction of all inter-thread data dependences.

Value Prediction: We can exploit *value prediction* by having the consumer of a potential dependence use a predicted value instead, as illustrated in Figure 5.1(c). After the epoch completes, it will compare the predicted value with the actual value; if the values differ, then the normal speculation recovery mechanism will be invoked to squash and restart the epoch with the correct value. We explore using value prediction as a replacement for both *speculation* and *synchronization*. In the former case (which we refer to later as "*memory value prediction*"), successful value prediction avoids the cost of recovery from an unsuccessful speculative load. In the latter case (which we refer to later as "*forwarded value prediction*"), successful prediction avoids the need to stall waiting for synchronization. Because the implementation issues and performance impact differ for these two cases, we evaluate them separately.

Silent Stores: An interesting program phenomenon that was recently discovered [5] is that many stores have no real side-effect since they overwrite memory with the same value that is already there. These stores are called *silent stores*, and we can exploit them when they occur to avoid failed speculation. Although one can view silent stores as a form of value prediction, the mechanism to exploit them is radically different from what is shown in Figure 5.1(c) since the changes occur with the *producer* of a communicated value, rather than the consumer.

Hardware-Inserted Dynamic Synchronization: In cases where the compiler decided to speculate that an ambiguous store/load dependence between speculative threads was not likely to occur, but where the dependence does in fact occur frequently and the communicated value is unpredictable, the best option would be to explicitly synchronize the threads (Figure 5.1(b)) to avoid the full cost of failed speculation. However, since the compiler did not recognize that such synchronization would be useful, another option is for the *hardware* to automatically switch from speculating to synchronizing when it dynamically detects such bad cases.

Reducing the Critical Forwarding Path: Once synchronization is introduced to explicitly forward values across epochs, it creates a dependence chain across the threads that may ultimately limit the parallel speedup. We

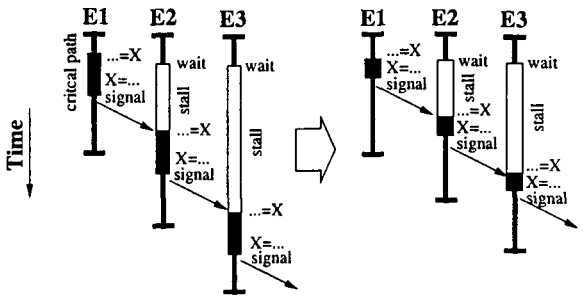


Figure 5.3: Reducing the critical forwarding path.

can potentially improve performance in such cases by using scheduling techniques to reduce the critical path between the first use and last definition of the dependent value, as illustrated in Figure 5.3. We implement both compiler and hardware methods for reducing the critical forwarding path.

5.1.2 Contributions and Overview

This chapter makes the following contributions. First, we evaluate a comprehensive set of techniques for enhancing value communication within a system that supports thread-level speculation, and demonstrate that many of them can result in significant performance gains. While we evaluate these techniques within the context of our own implementation of TLS, we expect to see similar trends within other TLS environments since the results are largely dependent on application behavior rather than the details of how speculation support is implemented. Second, and perhaps most importantly, we evaluate these techniques *after* the compiler has eliminated obvious data dependences and scheduled any critical forwarding paths, thereby removing the "easy" bottlenecks to achieving good performance. This leads us to very different conclusions than previous studies on exploiting value prediction within TLS [49, 59]. Third, we demonstrate the importance of throttling back value prediction to avoid the high cost of misprediction, and propose and evaluate techniques for focusing prediction on the dependences that matter most. Fourth, we present the first exploration of how *silent stores* can be exploited within TLS; compared with using traditional value prediction mechanisms to predict speculative memory loads, our silent stores approach yields comparable (if not better) performance while requiring considerably less hardware support. Finally, we evaluate two novel hardware techniques for enhancing the performance of synchronized dependences across speculative threads, but find mixed or disappointing results compared with what our compiler can do to optimize such cases in software.

The remainder of this chapter is organized as follows. In Section 5.2 we describe our approach to TLS support, including our hardware implementation and compiler infrastructure. We take a closer look at the potential for improving value communication in Section 5.3, and show that our compiler optimizations have a large impact. In Section 5.4 we investigate techniques for improving value prediction, and explore methods to improve synchronization. Finally, in Section 5.5 we evaluate the combination of all techniques, and summarize in Section 5.6.

5.2 Our Support for Thread-Level Speculation

This section describes the goals of our approach, how we implement support for TLS in hardware, our compiler support, and our experimental framework. While this study is within the context of our approach to TLS support [66, 67], the techniques that we suggest for improving value communication are applicable to other approaches as well.

5.2.1 Goals of Our Approach

Before we begin our investigation of value communication for TLS, it is important to understand the philosophy behind our approach. First and foremost, our goal is to parallelize general-purpose programs. Our scheme supports parallelization of scientific codes, but for now we focus on the more difficult problem of parallelizing integer applications. Second, we want to keep the hardware support simple and minimal: we avoid large structures that are

specialized for speculation, and preserve the performance of non-TLS workloads. Third, we take full advantage of the compiler which selects the regions of code to speculatively parallelize, eliminates data dependences where possible, and otherwise inserts synchronization and schedules the critical paths.

5.2.2 Underlying Hardware Support

Our scheme [66, 67] is applicable to shared-cache architectures, but for now we focus on single-chip multiprocessors where each processor has its own physically-private first-level data cache, connected to a unified second-level cache by a crossbar switch. TLS hardware support must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation. In our scheme we implement this support by using the data caches and an extended version of standard invalidation-based cache coherence.

In a nutshell, our coherence scheme works by tracking which cache lines have been speculatively loaded or modified, and piggybacking a sequence number on coherence messages to detect when an epoch has violated a data dependence. We buffer speculative modifications from regular memory by ensuring that speculatively-modified cache lines are not evicted[3] so that only committed, non-speculative modifications are visible to the rest of the memory hierarchy. We also provide support for *multiple writers*, where two epochs can each speculatively modify their own copy of the same cache line: the coherence mechanism uses the sequence numbers to properly combine the cache lines when they are committed.

5.2.3 Compiler Support

In contrast with hardware-only approaches to TLS, we rely on the compiler to define where and how to speculate. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [71], and performs the following phases when compiling an application to exploit TLS.

Deciding Where to Speculate: For this report, we focus solely on loops. With the help of automatically-gathered profile information, the compiler selects loops to maximize coverage while meeting heuristics for epoch size and loop trip counts: each loop must comprise at least 0.1% of overall execution time and have an average of at least 1.5 epochs per instance, as well as an average of at least 15 instructions per epoch. Once the key loops are selected, the compiler automatically applies loop unrolling to small loops to help amortize the overheads of speculative parallelization.

Transforming to Exploit TLS: Once speculative regions are chosen, the compiler inserts new TLS-specific instructions into the code that interact with the TLS hardware to create and manage the speculative threads (aka "epochs") [66]. We must also satisfy register dependences between speculative threads; to accomplish this, the compiler "forwards" register values between successive epochs by accessing a special portion of the stack called the *forwarding frame* which allows hardware to manage synchronization and communication for these values. Before the first use of a forwarded value, the compiler inserts a `wait` instruction, and then reads the value from the forwarding frame. After the last definition, the value is written back to the forwarding frame and a `signal` instruction allows the next epoch to proceed.

Optimization: Without optimization, execution can be unnecessarily serialized by synchronization (through `wait` and `signal` operations). A pathological case is a "`for`" loop in the `C` language where the loop counter is used at the beginning of the loop and then incremented at the end of the loop—if the loop counter is synchronized and forwarded then the loop will be serialized. However, scheduling can be used to move the `wait` and `signal` closer to each other, thereby reducing this critical path. Our compiler schedules these critical paths by first identifying the computation chain leading to each `signal`, and then using a dataflow analysis which extends the algorithm developed by Knoop [43] to schedule that code in the earliest safe location. We can do even better for any loop induction variable that is a linear function of the loop index; the scheduler hoists the associated code to the top of the epoch and computes that value locally from the loop index, avoiding any extra synchronization altogether. These optimizations have a large impact on performance, as we show later in Section 5.3.1.

Code Generation: Our compiler outputs `C` source code which encodes our new TLS instructions as in-line MIPS assembly code using `gcc`'s "`asm`" statements. This source code is then compiled with `gcc` v2.95.2 using the "`-O3`" flag to produce optimized, fully-functional MIPS binaries with TLS instructions.

[3] If a speculative cache line must be evicted, we simply cause speculation to fail for the corresponding epoch.

Table 5.1: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

Table 5.2: Benchmark statistics.

Application Name	Portion of Dynamic Execution Parallelized (Coverage)	Number of Unique Parallelized Regions	Average Epoch Size (dynamic insts)	Average Number of Epochs Per Dynamic Region Instance
BZIP2	98.1%	1	251.5	451596.0
CRAFTY	36.1%	34	30.8	1315.7
GZIP	70.4%	1	1307.0	2064.8
MCF	61.0%	9	206.2	198.9
PARSER	36.4%	41	271.1	19.4
PERLBMK	10.3%	10	65.1	2.4
VORTEX2K	12.7%	6	1994.3	3.4
VPR	80.1%	6	90.2	6.3
COMPRESS95	75.5%	7	188.2	68.4
GO	31.3%	40	2252.7	56.2
IJPEG	90.6%	23	1499.8	33.8
LI	17.0%	3	176.4	124.9
M88KSIM	56.5%	6	840.4	50.2
PERL	43.9%	4	137.3	2.2

5.2.4 Experimental Framework

We evaluate our support for TLS through detailed simulation. Our simulator models 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [82]. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 5.1. We simulate up to the first billion instructions[4] of SPECint95 and SPECint2000 benchmarks [23].[5]

5.3 A Closer Look at Improving Value Communication

In this section, we evaluate the impact of compiler optimization on performance and then show the potential for further improvement by hardware techniques.

5.3.1 Impact of Compiler Optimization

We begin by analyzing the performance impact of our compiler on TLS execution. Table 5.2 shows some statistics on our benchmarks. We observe that our *coverage* (i.e. the portion of dynamic execution time that has been parallelized using TLS) is reasonably good for most benchmarks: 51.4% on average, and as high as 98.1%. Figure 5.4 shows

[4] Since the sequential and TLS versions of each benchmark are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code.

[5] At the time of publication, our infrastructure could not yet handle GCC, TWOLF, GAP, nor EON.

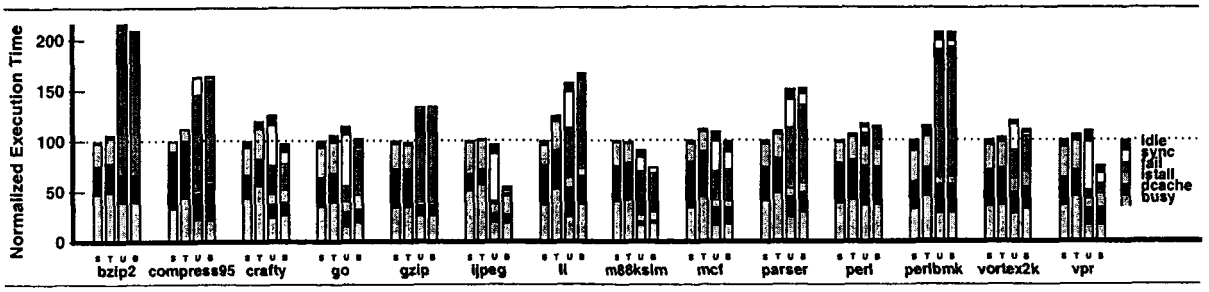


Figure 5.4: Performance impact of our TLS compiler. For each experiment, we show normalized region execution time scaled to the number of processors multiplied by the number of cycles (smaller is better). *S* is the sequential version, *T* is the TLS version run sequentially. There are two versions of TLS code run in parallel: *U* and *B* are without and with compiler scheduling of the critical forwarding path, respectively. Each bar shows a detailed breakdown of how time is being spent.

the performance on a single-chip, 4-processor multiprocessor. For each application in Figure 5.4, the leftmost bar (*S*) is the original sequential version of the code, and the next bar (*T*) is the TLS version of the code run on a single processor. For each experiment, we show region execution time normalized to the sequential case (*S*); hence bars that are less than 100 are speeding up, and bars that are larger than 100 are slowing down, relative to the sequential version. Comparing the TLS version run sequentially (*T*) with the original sequential version (*S*) isolates the overhead of TLS transformation. In all cases, this is roughly 10%. When we run the TLS code in parallel, it must overcome this overhead in order to achieve an overall speedup.

Each bar in Figure 5.4 is broken down into six segments explaining what happened during all potential graduation slots.[6] The *fail* segment represents all slots wasted on failed thread-level speculation, and the remaining five segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *dcache* segment is the number of non-graduating slots attributed to data cache misses; the *sync* portion represents slots spent waiting for synchronization for a forwarded location; the *istall* segment is all other slots where instructions cannot graduate; the *idle* segment represents slots where the reorder buffer is empty.

We consider two versions of TLS code running in parallel: the *B* case includes all of the compiler optimizations described earlier in Section 5.2.3, and the *U* case is the same minus the aggressive compiler scheduling to reduce the critical forwarding path. In nearly every case, the "unoptimized"[7] version (*U*) slows down with respect to the sequential version. The additional impediments include decreased data cache locality, synchronization, and failed speculation. Many benchmarks spend a significant amount of time synchronizing on forwarded values (as shown by the *sync* portion). Some benchmarks suffer from non-negligible *idle* segments; in general, this indicates load imbalance and in many cases it is due to regions that have fewer epochs than there are processors. If the compiler optimizes forwarded values by removing dependences due to certain loop induction variables and scheduling the critical path (*B*, our baseline), we observe that the performance of several benchmarks (CRAFTY, GO, IJPEG, M88KSIM, MCF, VORTEX2K, and VPR) improves substantially through decreased synchronization (*sync*), indicating that this is a crucial optimization.

5.3.2 The Potential for Further Improvement by Hardware

To illustrate that the performance of many of our benchmarks is limited by the efficiency of value communication, we show in Figure 5.5 the impact of ideal prediction on performance. First, in the *F* experiment we see the impact of perfect prediction of forwarded values. In effect, this means that there will be no time spent waiting for synchronization of forwarded values. Most benchmarks improve slightly, while M88KSIM, MCF, and PARSER show a substantial improvement: this makes sense since the baseline experiment (*B*) shows these benchmarks to be somewhat limited by synchronization. All benchmarks except for IJPEG and VPR suffer from a significant amount of failed speculation in the *B* and *F* experiments.

[6] The number of graduation slots is the product of: (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors.

[7] Note that the "unoptimized" case still includes the `gcc "-O3"` flag, and is optimized in every way except for the aggressive critical forwarding path scheduling.

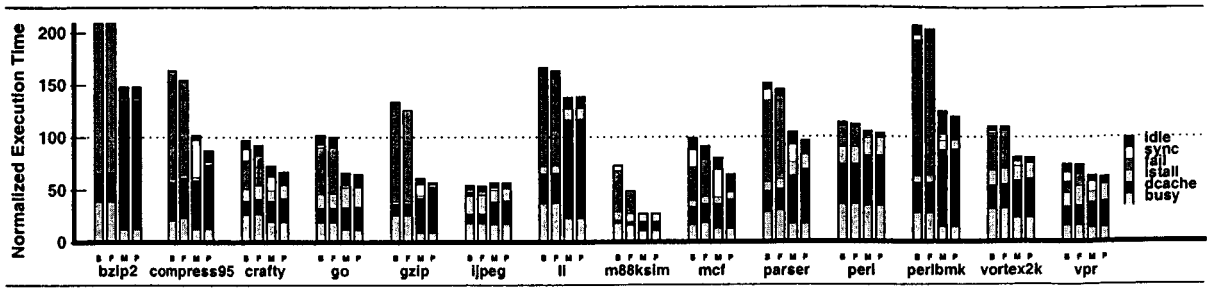


Figure 5.5: Potential for improved value communication. For each experiment, we show normalized region execution time (smaller is better). *B* is the baseline TLS version run speculatively in parallel, *M* shows perfect prediction of memory values, *F* shows perfect prediction of forwarded values, and *P* shows perfect prediction of both forwarded and memory values.

In the *M* experiment, we measure the impact of perfect memory value prediction, which means that no epoch will suffer from failed speculation. In this case, we see a great improvement in most benchmarks. CRAFTY, MCF, and PARSER show a significant synchronization portion for the *M* experiment, indicating that synchronization is still a limiting factor for these benchmarks.

Finally, in the *P* experiment we evaluate the impact of perfect prediction of both memory and forwarded value values. In this case, MCF and PARSER show a significant benefit compared with perfect memory value prediction alone, while the other benchmarks show only modest improvements. Evidently, avoiding failed speculation is the main bottleneck to good performance, while improving synchronization may still be important for some benchmarks. Also, if we cannot fully eliminate failed speculation then improving synchronization will still be important. Note that BZIP2, LI, PERL, and PERLBMK do not speed up, even with perfect prediction of memory and forwarded values. For these four benchmarks, the decreased data cache locality of executing on four processors is the limiting factor. IJPEG will not speed up further even under perfect prediction of both forwarded and memory values (*P*).

5.4 Evaluation of Techniques for Improving Value Communication

In this section we first focus on the benefits of predicting values for TLS: we describe the issues related to value prediction in the midst of speculation, describe how to predict memory values and how to predict forwarded values, and then discuss how to apply the technique of optimizing silent stores. Then, we focus on improving synchronization and automatically applying synchronization when speculation and prediction are ineffective.

5.4.1 Techniques for When Prediction Is Best

Value prediction in the context of a uniprocessor is fairly well understood [34, 45, 64, 78], while value prediction for thread-speculative architectures is relatively new. Gonzalez *et al.* [49] evaluated the potential for value prediction when speculating at a thread-level on the innermost loops from SPECint95 [23] benchmarks, and concluded that predicting synchronized (forwarded) register dependences provided the greatest benefit, and that predicting memory values did not offer much additional benefit. The opposite is true of our results for two reasons. First, our compiler has correctly scheduled easily-predictable but frequently-synchronized loop-induction variables so that they cannot cause dependence violations, and has also scheduled the code paths of forwarded values to minimize the impact of synchronization on performance. Second, we have selected much larger regions of code to speculate on, resulting in a greater number of memory dependences between threads. Concurrent with our work, Cintra *et al.* [20] investigated the impact of value prediction after the compiler has optimized loop induction variables in floating-point applications. Several other works evaluate the impact of value prediction without such compiler optimization. Oplinger *et al.* [59] evaluate the potential benefits to TLS of memory, register, and procedure return value prediction, and Akkary *et al.* [3] and Rotenberg *et al.* [63] also describe designs that include value prediction.

Predicting values for TLS has similar issues to predicting values in the midst of branch speculation, but at a larger scale. With branch speculation, we do not want to update the predictor for loads on the mispredicted path. Also, when a value is mispredicted we need only squash a relatively small number of instructions, so the cost of misprediction is

Table 5.3: Memory Value Prediction statistics.

Application	Avg. Exposed Loads per Epoch	Incorrect	Correct	Not Confi dent
BZIP2	9.5	0.2%	63.4%	36.3%
CRAFTY	4.5	3.0%	48.6%	48.3%
GZIP	66.6	1.4%	52.8%	45.7%
MCF	2.5	1.7%	34.9%	63.3%
PARSER	3.6	3.2%	48.7%	48.0%
PERLBMK	1.6	0.9%	17.9%	81.0%
VORTEX2K	25.4	2.8%	64.9%	32.2%
VPR	6.3	3.6%	49.8%	46.4%
COMPRESS95	12.0	0.3%	31.8%	67.9%
GO	7.0	2.5%	41.2%	56.2%
IJPEG	4.4	1.6%	35.4%	62.8%
LI	2.3	2.1%	50.8%	46.9%
M88KSIM	7.5	1.2%	90.9%	7.7%
PERL	12.3	1.1%	79.7%	19.1%

not large. Similarly, in TLS we only want to update the predictor for values predicted in successful epochs, but this will require either a larger amount of buffering or the ability to back-up and restore the state of the value predictor. Furthermore, the cost of a misprediction is high for TLS: the entire epoch must be re-executed if a value is mispredicted because a prediction cannot be verified until the end of the epoch when all modifications by previous epochs have been made visible.[8] Finally, for TLS we require that each epoch has a logically-separate value predictor. For SMT or other shared-pipeline speculation scheme, this does not mean that each requires a physically separate value predictor, but that the prediction entries must be kept separate by incorporating the epoch context identifier into the indexing function. This is necessary since multiple epochs may need to simultaneously predict different versions of the same location.

For this section, we model an aggressive hybrid predictor that combines a 1K×3-entry context predictor with a 1K-entry stride predictor, using 2-bit, up/down, saturating confidence counters to select between the two predictors. We found that the number of mispredictions can be minimized by simply predicting only when the prediction confidence is at the maximum value. Finding the smallest and simplest predictor that produces good results is beyond the scope of this section. It is important to note that we also model misprediction by re-executing any epoch that has used a mispredicted-value. A misprediction is not detected until the end of the epoch when the prediction is verified.

Memory Value Prediction

One potential way to eliminate data dependence violations between speculative threads is through the prediction of memory values. But which loads should we predict? A simple approach would be to predict every load for which the predictor is confident. Previous work [11, 30] shows that focusing prediction on critical path instructions is important for uniprocessor value prediction when modeling realistic misprediction penalties. Similarly, the cost of misprediction in TLS is very high, so we instead want to focus only on the loads that can potentially cause misspeculation. Fortunately, this information is available from the speculative cache line state: only loads that are *exposed*[9] can cause speculation to fail. Since our scheme tracks which words have been speculatively-modified in each cache line, we can decide whether a given load is exposed.

Table 5.3 shows some statistics for the prediction of exposed loads using the predictor described above. M88KSIM and PERL are quite predictable, while the remaining benchmarks also provide a significant fraction of correct predictions. We see that the amount of misprediction is quite small—fewer than 4% of predictions are incorrect for all benchmarks. Hence we expect memory value prediction to work well.

In Figure 5.6, the *E* experiment shows the impact of predicting all exposed loads for which the predictor is confident. In almost every case, performance is worse due to an increased amount of failed speculation caused by misprediction. The problem is that it only takes a single misprediction to cause speculation to fail.

Rather than predict all exposed loads, we can be more selective by only predicting loads that are likely to cause dependence violations. We can track these loads with the following two devices. First, we keep a 16-entry table (called the *exposed load table*) that is indexed by the cache line tag, and stores the PC of the corresponding exposed load.

[8] Some schemes support selective-squashing of instructions that have used a mispredicted value [3], but this requires a large amount of buffering.

[9] A load that is not proceeded in the same epoch by a store to the same location is considered to be exposed [2].

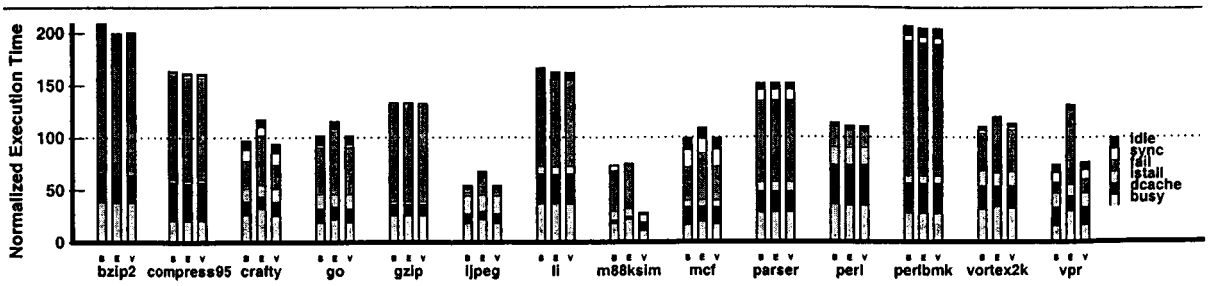


Figure 5.6: Performance with memory value prediction. *B* is the baseline experiment, *E* predicts all *exposed* loads, and *V* only predicts loads that have caused violations.

Table 5.4: Forwarded Value Prediction statistics.

Application	Incorrect	Correct	Not Confi dent
BZIP2	0.0%	0.0%	0.0%
CRAFTY	5.5%	24.6%	69.7%
GZIP	0.2%	98.0%	1.6%
MCF	2.5%	48.5%	48.9%
PARSER	2.8%	11.6%	85.5%
PERLBMK	2.9%	61.7%	35.2%
VORTEX2K	2.2%	81.9%	15.7%
VPR	2.8%	26.4%	70.7%
COMPRESS95	3.7%	31.2%	65.1%
GO	3.7%	28.3%	67.9%
IJPEG	5.8%	72.4%	21.6%
LI	1.0%	18.7%	80.1%
M88KSIM	5.4%	91.0%	3.4%
PERL	1.5%	91.4%	7.0%

Subsequent exposed loads simply overwrite the appropriate entry—hence we track the most recent exposed loads. Second, whenever a dependence violation occurs it is associated with a cache line, so we can use the cache line tag to index the exposed load table and retrieve the PC of the offending load. Hence we can keep a list of load PCs which have caused violations (the *violating loads list*), and can now use this list to decide which loads we should predict. In Figure 5.6, the *V* experiment shows the impact of predicting only loads that have caused violations, as given by the violating loads list. Compared with the baseline *B*, we see that every benchmark either improves slightly or at least remains unchanged, except for VORTEX2K which degrades slightly, and M88KSIM which improves significantly by eliminating much failed speculation. Hence with proper throttling, memory value prediction can be used to improve the performance of some applications.

Having explored value prediction for the sake of avoiding failed speculation, we now turn our attention to using value prediction to mitigate the performance impact of explicit synchronization.

Prediction of Forwarded Values

Recall that forwarded values are those that are frequently modified and so they are synchronized between epochs by the compiler. Just as we did with memory values, we can also predict forwarded values. However, while we predict memory values to decrease the amount of failed speculation, we predict forwarded values to decrease the amount of time spent synchronizing. Table 5.4 shows some statistics for the prediction of forwarded values. We see that the fraction of incorrect predictions is somewhat higher than for memory values, and the fraction of correct predictions is not as high.

In Figure 5.7, the *F* experiment shows the impact of predicting forwarded values: MCF improves by 4.2%, GZIP by 6.0%, and M88KSIM by 43.6%. The remaining benchmarks are not greatly affected, except for CRAFTY and VPR which become slightly worse due to mispredictions. In an attempt to remedy this problem, we applied a similar technique to that used in memory value prediction: we track which forwarded loads that cause the pipeline to stall

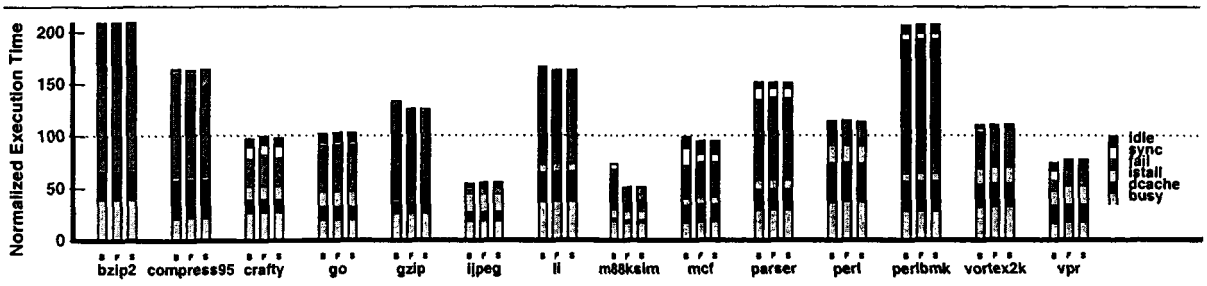


Figure 5.7: Performance of forwarded value prediction. *B* is the baseline experiment, *F* predicts all forwarded values *S* predicts forwarded values that have caused stalls.

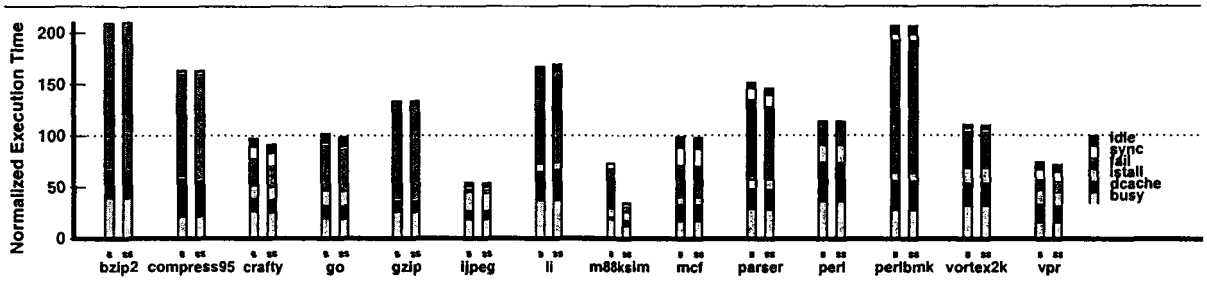


Figure 5.8: Performance of silent stores optimization. *B* is the baseline experiment, and *SS* optimizes silent stores.

waiting for synchronization, and only predict those values. The *S* bars shows the results of this experiment, which maintains the performance of the *F* experiment in every benchmark, and solved the problem for CRAFTY but not for VPR. In summary, the prediction of forwarded values is an effective way to reduce the amount of time spent synchronizing for some applications.

Silent Stores

Often, a store does not actually modify the value of its target location. In other words, the value of the location before the store is the same as the value of the location after the store. This occurrence is known as a *silent store* [5], and was first exploited to reduce coherence traffic. A store that is expected to be silent is replaced with a load, and the loaded value is compared with the value to be stored. If they are not the same, then the store is executed after all, otherwise we save the coherence traffic of gaining exclusive access to the cache line and eliminate future update traffic. This same technique can be applied to TLS to avoid data dependence violations so that a dependent store-load pair can be made independent if the store is silent.

In Table 5.5 we see that silent stores are abundant, ranging from 4% to 80% of all dynamic non-stack stores within speculative regions. However, what matters is whether the stores which cause dependence violations are silent. Figure 5.8 shows that optimizing silent stores results in a slight improvement for most benchmarks, and a large improvement in M88KSIM; only LI performs slightly worse when optimizing silent stores. Compared with using value prediction to avoid potential memory dependences (as we explored earlier in Section 5.4.1), this silent stores approach yields similar if not better performance, but requires significantly less hardware support (e.g., no value predictor is needed). Hence this appears to be a very attractive technique for enhancing TLS performance.

5.4.2 Techniques for When Synchronization Is Best

We now turn focus on the scenarios when synchronization is the right thing to do. We investigate techniques for dynamic synchronization of dependences, and prioritization of the critical path.

Table 5.5: Percent of Dynamic, Non-Stack Stores That Are Silent.

Application	Dynamic, Non-Stack Silent Stores
BZIP2	11%
CRAFTY	16%
GZIP	4%
MCF	19%
PARSER	12%
PERLBMK	7%
VORTEX2K	84%
VPR	26%
COMPRESS95	80%
GO	16%
IJPEG	31%
LI	19%
M88KSIM	57%
PERL	36%

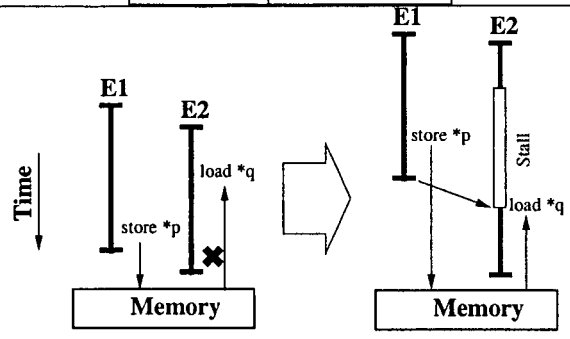


Figure 5.9: Dynamic synchronization, which avoids failed speculation (left) by stalling the appropriate load until the previous epoch completes (right).

Hardware-Inserted Dynamic Synchronization

For many of our benchmarks, failed speculation is a significant performance limitation; and as we observed in Section 5.4.1, prediction alone cannot eliminate all dependence violations. For dependences with unpredictable values that occur frequently, the only remaining alternative is to synchronize. Our compiler has already inserted synchronization for local variables. Still many dependences remain, as demonstrated in Section 5.3.2, which can be synchronized dynamically by hardware.

Dynamic synchronization has been applied to both uniprocessor and multiprocessor domains. Chrysos *et al.* [17] present a design for dynamically synchronizing dependent store-load pairs within the context of an out-of-order issue uniprocessor pipeline, and Moshovos *et al.* [54] investigate the dynamic synchronization of dependent store-load pairs in the context of the Multiscalar architecture [33, 65].

Both of these works differ from ours because they have the ability to forward a value directly from the store to the load in a dynamically-synchronized store-load pair: this is trivial in a uniprocessor since the store and load issue from the same pipeline; for a multiprocessor like the Multiscalar, this requires that the memory location in question is implicitly forwarded from the producer to the consumer—functionality that is provided by the Multiscalar's *address-resolution buffer* [33]. Our scheme does not provide this support because it would require the memory coherence protocol to perform complex version management.

Figure 5.9 illustrates how we dynamically synchronize. When a load is likely to cause a dependence violation, we can prevent speculation from failing by instead stalling the load until the previous epoch is complete: at that point, all modifications by previous epochs will be visible and the load can safely issue. We can use the *violating loads list* described in Section 5.4.1 to identify the loads most likely to cause a violation that should therefore be synchronized.

In Figure 5.10, experiment *D* shows the performance of dynamic synchronization where we have synchronized every load in the violating loads list. By inspecting both result graphs, we see that failed speculation has been replaced with synchronization as expected, resulting in improved performance for 10 of the 14 benchmarks. However, CRAFTY, GZIP, and VPR are now over-synchronized: we have unwittingly replaced successful speculation with synchronization

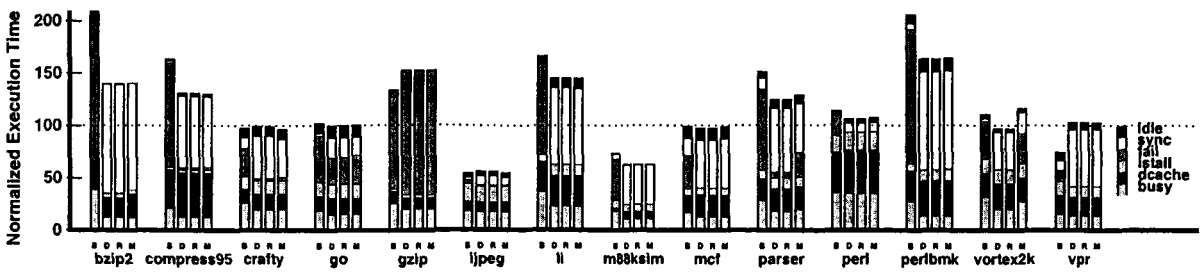
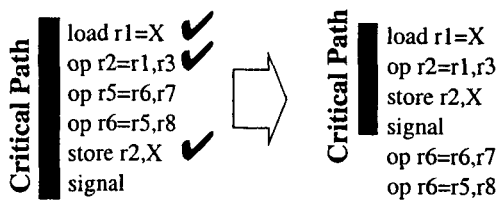


Figure 5.10: Performance of dynamic synchronization. *B* is the baseline experiment, *D* automatically synchronizes all violating loads, *R* builds on *D* by periodically resetting the violating loads list, and *M* builds on *R* by requiring a load to have caused at least 4 violations since the last reset before synchronizing it.

(a) *Prioritization*



(b) *Statistics*

Application	Issued Insts That Are High Priority and Issued Early	Improvement in Avg. Start-to-Signal Time (cycles)		
		Unprioritized	Prioritized	Speedup
BZIP2	0.0%	0.0	0.0	0.0
CRAFTY	6.8%	118.5	117.7	0.99
GZIP	3.6%	1622.6	1636.2	1.00
MCF	9.9%	86.7	80.2	0.92
PARSER	9.7%	110.9	105.6	0.95
PERLBMK	18.1%	45.5	44.0	0.96
VORTEX2K	3.6%	304.8	290.4	0.95
VPR	4.7%	81.8	79.7	0.97
COMPRESS95	7.1%	136.7	135.9	1.01
GO	12.9%	70.2	70.4	1.00
IJPEG	11.2%	28.1	26.0	0.92
LI	27.5%	37.4	33.1	0.88
M88KSIM	9.1%	212.8	218.4	1.02
PERL	16.6%	42.4	44.5	1.04

Figure 5.11: Prioritization of the critical path. We show (a) our algorithm, where we mark the instructions on the input chain of the critical store and the pipeline's issue logic gives them high priority; (b) some statistics, namely the fraction of issued instructions that are given high priority by our algorithm and issue early, and also the improvement in number of cycles from the start of the epoch until each signal.

as well. In an attempt to mitigate this effect, we periodically reset the violating loads list in experiment *R*, and build on that in experiment *M* by requiring that a load be responsible for at least 4 violations since the last reset before synchronizing. The *M* experiment solves the problem for CRAFTY but not for VPR, and performance is degraded for PARSER and VORTEX2K. Overall, this technique has a greater benefit than cost (an average improvement of 9%), and is a promising technique for improving the performance of TLS.

Prioritizing the Critical Path

In Section 5.4.1 we observed that even after aggressive prediction of forwarded values, synchronization is still an impediment to good speedup for some benchmarks. We call the instructions between the first use and the last definition

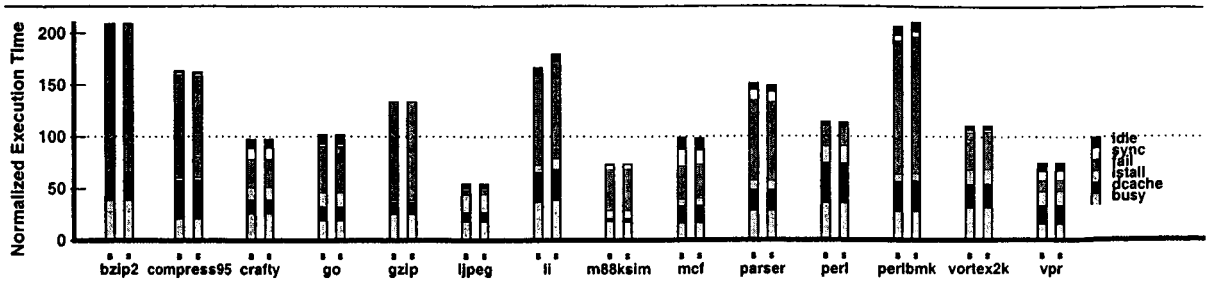


Figure 5.12: Performance impact of prioritizing the critical path: *B* is the baseline experiment, and *S* prioritizes the critical path.

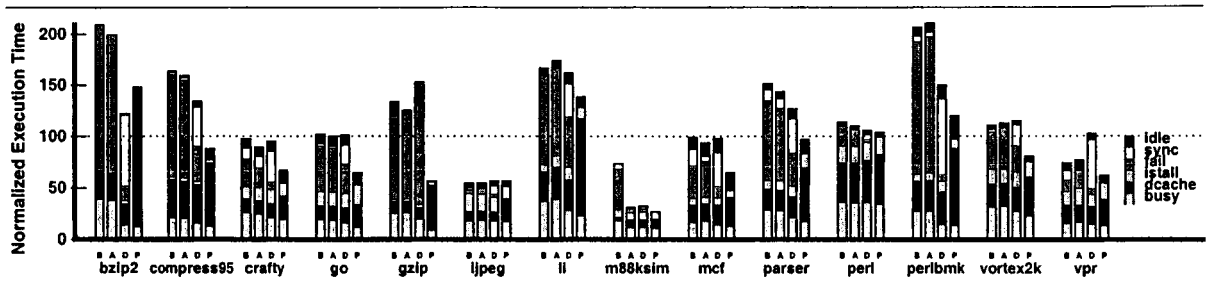


Figure 5.13: Performance of all techniques combined. *B* is the baseline experiment, *A* performs all optimizations except dynamic synchronization, *D* performs all optimizations, and *P* is the perfect prediction result from Section 5.3.2 for comparison.

of a forwarded value the *critical path*. A possibility for improving performance when it is not possible to eliminate synchronization is to instead prioritize instructions to help reduce the size of the critical forwarding path. Our compiler already performs this optimization to the best of its ability, but there may be more that can be done dynamically by hardware at run-time.

Our hardware prioritization algorithm works as shown in Figure 5.11(a). We mark all instructions with registers on the input-chain of the critical store. We also track the critical path through memory, so that a critical load also depends on the store which produced the value for the given memory location. Ideally, we would also mark any instructions on the input-chain of an unpredictable conditional branch as being on the critical path, but this beyond the scope of this section. The pipeline issue logic then gives priority to marked instructions so that the associated signal may be issued as early as possible. This algorithm could be implemented using techniques described by Fields *et al.* [30], but for now we focus on the potential impact.

The impact of prioritizing the critical path is shown in Figure 5.12. Note that we model a 128-entry reorder buffer (see Table 5.1), so the issue logic has significant opportunity to reorder prioritized instructions. Despite this fact, all benchmarks remain relatively unchanged, while MCF and PARSER improve slightly, and the performance of LI and PERLBMK degrades slightly. To clarify whether our prioritization has had any impact, Figure 5.11(b) shows the fraction of issued instructions that are given high priority by our algorithm and also issue early. This is between 4% and 28% of issued instructions, an average of 10.8% across all benchmarks with the exception of BZIP2 (which does not have forwarded values). Figure 5.11(b) also shows the change in the average number of cycles from the start of an epoch to the issue of each signal, for which the results are mixed: COMPRESS95, M88KSIM, and PERL have improved somewhat, GZIP and GO are unchanged, and the remaining benchmarks have slowed-down slightly. Given the potential complexity for implementing this technique and the resulting uncompelling performance, we do not advocate the use of this technique.

5.5 Combining the Techniques

In this Section, we evaluate the impact of all of our techniques combined. Most techniques are orthogonal in their operation with the exception of memory value prediction and dynamic synchronization: we only want to dynamically synchronize on memory values that are unpredictable. This cooperative behavior is implemented by having the dynamic synchronization logic check the prediction confidence for the load in question, and synchronizing only when confidence is low.

Figure 5.13 shows the performance of all techniques combined, where *A* performs all optimizations except dynamic synchronization, *D* performs all optimizations, and *P* is the perfect prediction result from Section 5.3.2 for comparison. We achieve very close to the ideal speedup for M88KSIM, and we have improved CRAFTY and MCF significantly. After all of our optimizations, we observe that failed speculation remains a problem for many benchmarks. For BZIP2 the *D* experiment shows how some techniques can be complementary since its performance is better than that of any one technique alone.[10] Since including dynamic synchronization (*D*) degrades performance for more than half of the benchmarks, we do not advocate this technique in its current form.

5.6 Summary

We have shown that improving value communication in TLS can yield large performance benefits, and examined the techniques for taking advantage of this fact. Our analysis provides several important lessons. First, we discovered that prediction cannot be applied liberally when the cost of misprediction is high: predictors must be throttled to target only those dependences that limit performance. We observed that silent stores are prevalent, and squashing them can greatly improve the performance of TLS execution. We found that dynamic synchronization improves performance for many applications but can degrade performance for others—this technique requires further throttling before it can be applied liberally. We also found that hardware prioritization to reduce the critical forwarding path does not work well, even though a significant number of instructions can be reordered. Finally, we have shown that compiler transformations can impact conclusions about hardware TLS support, and demonstrated that the compiler can be quite effective at improving the performance of TLS.

[10] The *D* bar out-performs the perfect prediction estimate (*P*) because that estimate does not account for the coherence traffi c savings of memory value prediction and silent stores—only for the savings in failed speculation.

Chapter 6

Compiler Optimization of Scalar Value Communication

6.1 Introduction

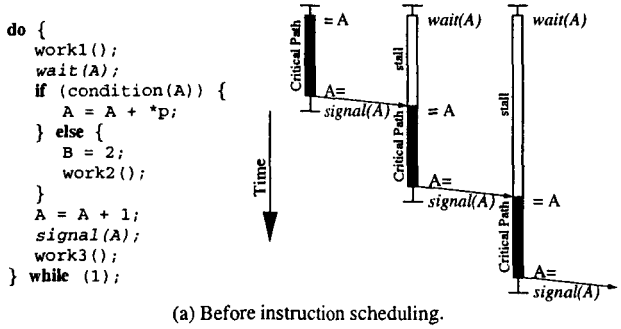
While recent research has investigated hardware optimization for TLS [20, 44, 51, 69, 55], there has been relatively little work on compiler optimization in this area. One potential opportunity for optimization focuses on data dependences between speculative threads that occur frequently: if the compiler is able to identify the source and the destination of a frequent inter-thread data dependence, then it is beneficial to insert synchronization and forward that value explicitly to avoid failed speculation. Figure 6.1(a) shows an example loop that the compiler has speculatively parallelized by partitioning the loop into speculative threads (aka *epochs*). Since the variable `A` is read and written in every iteration, the compiler decides to synchronize and forward `A` by inserting a `wait` operation before the first *use* of `A`, and a `signal` operation after the last *definition* of `A`—we describe, implement, and evaluate this algorithm in Section 6.3. The synchronization results in the partially-parallel execution shown in Figure 6.1(a), where each epoch stalls until the value of `A` is produced by the previous epoch. The flow of the value of `A` between epochs serializes the parallel execution, and so we refer to it as a *critical forwarding path*. In the next section, we show that the overall performance of speculation is limited by the size of this critical forwarding path. [1]

6.1.1 The Importance of Reducing the Critical Forwarding Path

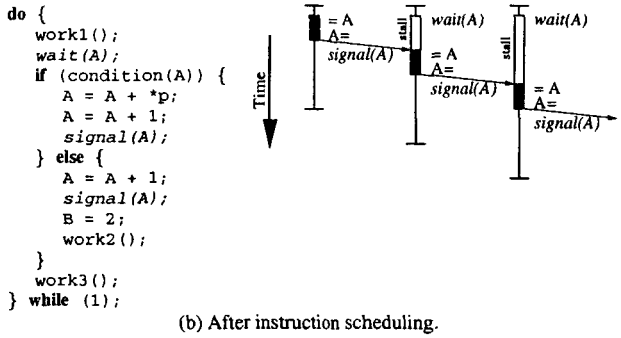
Although synchronization is better than speculation for a data dependence that occurs frequently, the resulting serialization can still limit performance. In fact, the performance of many applications that exploit TLS is limited by the critical forwarding path. Figure 6.2 shows the potential impact of reducing the critical forwarding path on a four-processor chip multiprocessor—we will explain the details of this experiment later in Section 6.2. The *U* bars show the unscheduled TLS version of each application run speculatively in parallel. Each bar is normalized to the execution time of the original sequential version, such that bars less than 100 are speeding up. The best we can possibly do with scheduling is to eliminate the critical forwarding path altogether. We measure this ideal behavior with a model that can perfectly predict all forwarded values such that there is no synchronization (*P*). We see that for most applications, removing the synchronization bottleneck results in great performance improvements.

What can the compiler do to shrink the critical forwarding path? The key idea is to reduce the number of instructions between each `wait/signal` pair. However, this becomes more difficult in the presence of conditional control flow. Figure 6.1(b) shows the example loop after the compiler has scheduled the code to reduce the critical forwarding path. The scheduling algorithm has duplicated the computation of `A=A+1` as well as the `signal` and moved them into the conditional structure. If the condition on `A` is rarely true, then less work will be performed before reaching each `signal` (by deferring the computation of `B=2` and `work2()`). As shown in the figure, this reduces the stall time for each epoch, thereby improving overall execution time. We describe an algorithm for reducing the critical path in Section 6.4.1.

[1] Results shown in this section can also be found in our previous publication. [84]



(a) Before instruction scheduling.



(b) After instruction scheduling.

Figure 6.1: Impact of scheduling on the critical forwarding path.

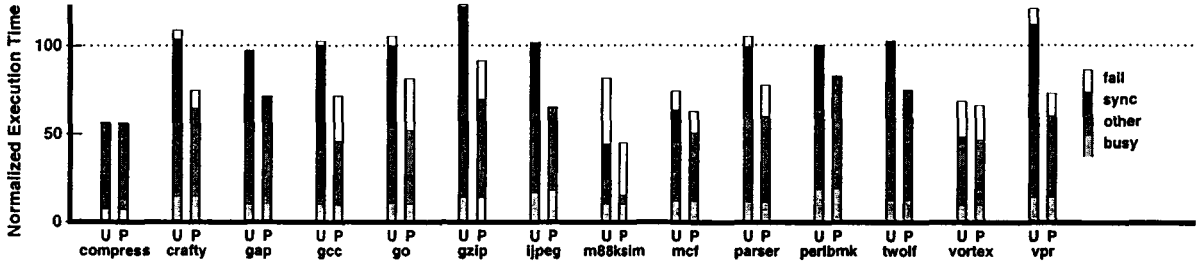


Figure 6.2: Potential impact of reducing the critical forwarding path. For each benchmark we show execution time on four processors for the speculatively parallelized regions of code normalized to that of the original sequential versions. *U* is the unscheduled speculative version and *P* shows the impact of perfect prediction of forwarded values.

6.1.2 More Aggressive Instruction Scheduling

All of the transformations that we have described so far will preserve the control and data dependences within each epoch: the transformed code will perform the same operations as the original, but possibly reordered within each control structure and between ambiguous data dependences. However, it is potentially beneficial to move code past control and data dependences [12, 31, 35, 58] to further reduce the critical forwarding path. For example, if a certain path is executed more frequently than alternative paths, then it is advantageous to speculatively schedule the critical forwarding path to exploit this fact. To illustrate, if the *else* clause is more frequently executed than the *then* clause in Figure 6.1(b), we could schedule "`A=A+1; signal(A);`" from the *else* clause above the *if* structure to further shrink the critical forwarding path in the common case. Thus our new schedule involves control speculation, and requires the ability to recover whenever our speculation is incorrect. Similarly, we can schedule code from the critical forwarding path past ambiguous data dependences, given the additional hardware support to detect when such speculation has failed. We describe and evaluate schemes for scheduling the critical forwarding path using intra-epoch control speculation and data dependence speculation in Section 6.4.3.

6.1.3 Related Work

Parallelization of a loop where the compiler synchronizes a loop-carried data dependence is known as a *DOACROSS* [26, 60] parallelization and has been exploited in previous work [13, 52, 87]. All schemes for TLS support include some form of DOACROSS synchronization, although few use the compiler to optimize this aspect of speculative execution.

The most relevant related work is the Wisconsin Multiscalar [32, 65, 77] compiler, which performs synchronization and scheduling for register values [77]. (The Multiscalar effort also evaluated hardware support for automatically detecting and synchronizing data dependences [55].) The Multiscalar scheduler was designed with Multiscalar tasks in mind, and these usually consist of a few basic blocks that do not contain procedure calls or loops. In contrast, our speculative threads (aka *epochs*) are much larger on average than Multiscalar tasks and contain complex control flow. This inspired the dataflow-based scheduler presented in this chapter, which can move instructions past inner loops and procedure calls. The Multiscalar compiler does not schedule code beyond the point within a task where it is no longer critical, as determined by a simplified machine model; in contrast, because we believe that accurate determination of this point at compile-time is extremely difficult, we schedule producer instructions as early as possible. Another difference is that our more general approach to scheduling handles loop induction variables automatically (by scheduling them at the top of the loop), rather than having to treat them as a special case. A final difference is that we evaluate the benefits of speculatively scheduling code past control and data dependences (as discussed later in Section 6.4.3). We modified our scheduler to mimic the Multiscalar scheduler, and we contrast the performance impact of both approaches later in Section 6.5.3.

Other schemes for TLS hardware support provide the means to synchronize and forward values between speculative threads but do not use the compiler to optimize loop-induction variables or synchronize frequent dependences [3, 39, 40, 48], while others provide such support but do not schedule instructions to reduce the critical forwarding path [20, 73]. Research on TLS hardware support has shown the importance of the critical forwarding path and how the prediction of forwarded values may be used to increase parallelism [51, 69]; it also showed that hardware is ineffective at improving performance by scheduling the critical forwarding path [69]. Other hardware techniques for improving the efficiency of speculation include prediction of loads to memory, dynamic synchronization, and squashing of *silent stores* (which overwrite memory with the same value that is already there) [3, 20, 48, 69].

Concurrent with our work, Zilles and Sohi [88] recently proposed decomposing a program into speculative threads by having a master thread execute a *distilled* version of the program that orchestrates and predicts values for slave threads. In this scheme, values are precomputed by the master thread and distributed to the slave threads (as opposed to being updated and forwarded between consecutive speculative threads). A potential advantage of this master/slave approach is that it effectively removes interprocessor communication from the critical forwarding path. We note that the scheduling techniques that we present later in this chapter could potentially be applied to the distilled code in the master thread.

Our algorithm for reducing the critical forwarding path builds upon previous dataflow approaches to code motion, namely *partial redundancy elimination* [43], path-sensitive dataflow analysis [41], and *hot paths* [4]. Previous work on speculative code motion to exploit a frequently executed path includes trace scheduling [31] and superblock scheduling [12]. There has also been work on aggressive load/store reordering where the runtime check and recovery are performed entirely in software [58] or through a hybrid hardware/software approach [35].

6.1.4 Contributions

In the context of thread-level speculation, this work makes the following contributions. First, we show that the critical forwarding path is a significant performance bottleneck for many applications. Second, we present novel dataflow scheduling algorithms for reducing the critical forwarding path, and show that scheduling loop induction variables and other scalars results in significant performance gains for most applications. Finally, we compare and contrast our compiler scheduling techniques with hardware techniques for optimizing the critical forwarding path [69].

6.2 Infrastructure for TLS

In this section we describe our compiler infrastructure and target hardware support for TLS, as well as our simulation infrastructure and experimental framework.

Table 6.1: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

6.2.1 Compiler Infrastructure

Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [71]. In addition to scheduling the critical forwarding path, our compiler also performs the tasks described below when automatically transforming a program to exploit TLS.

Deciding Where to Speculate: For this chapter, we focus solely on loops (at any nesting depth) as candidates for parallel execution (i.e. speculative *regions*), where loop iterations are the units of parallelism. Based on profile information, we first discarded loops which did not iterate, had a program coverage of less than 0.1%, executed fewer than 30 instructions per loop invocation, or had more than 16384 instructions per iteration, since these were unlikely to produce significant interesting parallelism. We then parallelized each of the remaining loops and profiled them using a simple in-order single instruction per cycle simulator. During this profiling we ignored the effects of all synchronization, thereby measuring an optimistic upper bound on performance—this assumes that the techniques in this chapter will remove any negative impact due to synchronization. The loops that performed best under these ideal conditions were then selected for use in this study. We found that some of the loops chosen based on our simple simulator performed poorly when run on a realistic simulator, since the simple simulator takes neither communication latency nor the cost of thread creation into account. We also show results for a subset of the loops that perform well—the details of the selection process are discussed below in Section 6.2.2.

Inserting TLS-Specific Instructions: Once speculative regions are chosen, the compiler inserts new TLS-specific instructions that interact with hardware to create and manage epochs. The compiler allocates forwarded variables on a special portion of the stack called the *forwarding frame* which supports the communication of values between epochs, and inserts `wait` and `signal` primitives according to the algorithms described in the remainder of this chapter. The `wait` and `signal` primitives combine synchronization with communication, acting as loads from and stores to the forwarding frame.

Generating Object Code: Our compiler outputs C source code which encodes our new TLS instructions as in-line MIPS assembly code using `gcc`'s "`asm`" statements. This source code is then compiled with `gcc` 2.95.2 using the "`-O3`" flag to produce optimized, fully-functional MIPS binaries with TLS instructions.

6.2.2 Region Selection

Since some of the regions selected using the process above performed poorly on our realistic simulator, we created a second set of regions that elides poor performers so that the performance of those regions with potential is not overwhelmed. Throughout this chapter we therefore evaluate performance on two sets of speculative regions for each application: (i) a set selected by the criteria given above (*"all regions"*), and (ii) a subset where we have pruned the regions for which speculative parallelization is ineffective despite the most aggressive compiler and hardware optimizations (*"good regions"*). Our results show that most of the pruned regions were not dominated by synchronization overhead, and hence they could not benefit from the techniques described in this chapter.

The GCC benchmark stands out as an exception: we pruned a few loops (with a combined program coverage of 4.4%) which were dominated by synchronization yet were not improved by our techniques. Some of the corresponding critical forwarding paths contained inner loops with a significant amount of computation, while others contained lengthy function calls—hence our scheduling techniques were unable to reorder the instructions effectively.

Table 6.2: Benchmark statistics.

Suite	Application Name	All Regions				Good Regions	
		Portion of Dynamic Execution Parallelized	Number of Unique Parallelized Regions	Average Epoch Size (dynamic insts)	Average Number of Epochs Per Dynamic Region Instance	Portion of Dynamic Execution Parallelized	Number of Unique Parallelized Regions
SpecINT2000	186.CRAFTY	28.7%	24	184.3	7.9	16.9%	8
	254.GAP	10.9%	1	94.0	4.8	10.9%	1
	176.GCC	43.2%	104	627.3	160.3	24.1%	57
	164.GZIP	7.9%	2	40.5	27229.8	2.5%	1
	181.MCF	49.9%	11	56.7	306.8	39.5%	3
	197.PARSER	58.1%	36	579.2	23644.5	10.8%	17
	253.PERLBMK	29.9%	10	363.9	405.0	16.3%	2
	300.TWOLF	26.1%	12	212.3	4.3	11.1%	3
	255.VORTEX	14.0%	6	4775.6	4.1	4.3%	4
	175.VPR	78.5%	5	236.9	5.8	65.3%	2
SpecINT95	129.COMPRESS	38.5%	1	125.0	863.0	38.5%	1
	099.GO	86.2%	73	1370.6	31.8	17.4%	24
	132.IJPEG	95.1%	21	245.4	95.8	60.7%	13
	124.M88KSIM	59.5%	6	790.1	60.3	54.6%	5

6.2.3 Underlying Hardware Support

TLS hardware support must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation, which we implement using the first-level data caches and an extended version of invalidation-based cache coherence [66, 67]. While we evaluate our compiler support on this specific implementation of TLS, we expect that our conclusions would be similar for other TLS hardware implementations [3, 20, 37, 39, 40, 44, 48, 59, 67, 73].

6.2.4 Experimental Framework

We evaluate our compilation techniques using a detailed machine model which simulates 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [82], but modernized to have a 128-entry reorder buffer. Each processor has its own physically private data and instruction caches, connected to a unified second level cache by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 6.1.

After skipping over the initialization phases, we simulate up to the first billion instructions of the applications described in Table 6.2. (Since the sequential and TLS versions of each application are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code.) We simulate all of the SPECint95 and SPECint2000 benchmarks [23] except for the following: 252.EON, which is written in C++ and therefore not handled by SUIF; 126.GCC, which is similar to 176.GCC; 147.VORTEX, which is identical to 255.VORTEX; and 256.BZIP2, 130.LI, and 134.PERL, the three of which have no loops that both comprise an interesting portion of execution and also are speculatively parallelizable by our current techniques.

6.3 Inserting Synchronization

This section presents a general algorithm for inserting synchronization to communicate values between epochs. In this chapter, we focus on the set of local (i.e. defined in the scope of the enclosing procedure) *communicating scalars*, which we define as the subset of local scalar variables that meet one of the following two criteria. First, any scalar in the intersection of the set of scalars with a downwards-exposed definition and an upwards-exposed use (i.e., the scalar is *live* between epochs). Second, any scalar that is live when the loop exits. In contrast with local scalars, global scalar and pointer references may be modified by calls to other procedures; synchronizing these memory references is addressed in Chaper 7.

For each synchronized scalar, the compiler inserts instructions that perform the synchronization and communicate the value. This is performed by the `wait` and `signal` instructions, each of which is associated with the scalar in question through an architected register. The `wait` instruction stalls execution until the value is produced by the previous epoch, which communicates that value through the `signal` instruction. For the first epoch of a speculatively-parallelized loop, the `wait` instruction does not stall (since there is no producer). In remainder of this section, we

describe where to place the synchronization instructions to ensure correct execution.

6.3.1 Constraints on Placement

The proper placement of `wait` and `signal` instructions can be described by a series of constraints which we describe here for a single synchronized scalar; the same constraints can be applied to each synchronized scalar individually. First, we want the last write to a scalar in an epoch to execute before the next epoch reads that scalar, regardless of the path of execution taken by either epoch. Hence we have the first two constraints:

1. a `wait` must occur before any use of the scalar on any path;
2. a `signal` must occur after the last definition of the scalar on any path.

In reality, if a signal was omitted on a given path then the waiting epoch could continue executing once all previous epochs were complete (rather than stalling indefinitely)—however, we ignore this to allow the placement algorithms to be symmetric. Hence, we have the additional constraint:

3. a `signal` must occur for each synchronized scalar on every possible path.

Given these first three constraints, a correct program can be created by trivially placing all `wait` instructions at the top of each epoch, and all `signal` instructions at the bottom of each epoch. However, such a transformation would completely serialize execution. To remedy this situation, we apply two additional constraints for the sake of improving performance:

4. each `wait` should be placed as late as possible;
5. each `signal` should be placed as early as possible.

6.3.2 Placement Algorithm

Intuitively, a placement algorithm for `wait` instructions would involve putting a `wait` for a scalar at the top of the epoch, and then pushing the `wait` downwards through the code. When a branch is encountered, the `wait` can be duplicated and pushed down on either side of the branch. The motion stops when a use of the scalar is encountered. For placement of `signal` instructions, the converse of this algorithm is used. Deciding which basic block should contain a `wait` or `signal` can be implemented as a dataflow analysis (described in detail below): within a basic block, the `wait` is placed directly above the first use of the scalar, and the `signal` is placed directly below the last definition of the scalar.

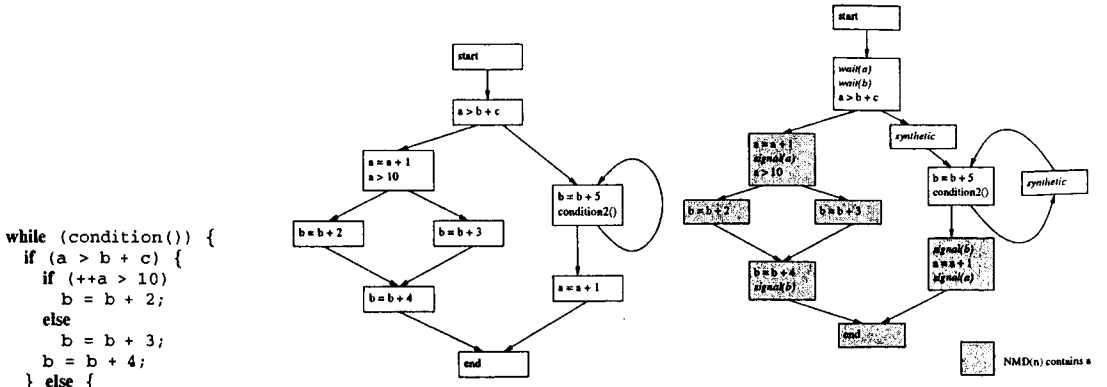
We now present a dataflow algorithm for placing `wait` and `signal` instructions in accordance with the above constraints. While we only show the algorithm for placing `signal` instructions, note that the converse of this algorithm is used to place `wait` instructions. (A proof of the correctness of this algorithm can be found in our technical report [85].)

We define our dataflow analysis over the set of communicating scalars V on the control flow graph $G = (N, E, s, e)$ of the epoch where N is the set of nodes which represent basic blocks, E is the set of edges, and s and e are the unique *start node* and *end node* of G (note that the start node and end node contain no code). Since *critical edges* (i.e. any edge connecting a node with more than one successor to a node with more than one predecessor) would make our analysis difficult, we break any such edges into two edges using *synthetic nodes* [43].

At each node $n \in N$ we define a predicate $LocalDef(n)$ to be the set of communicating scalars that are defined at n. Since the `signal` instruction that forwards the value of $v \in V$ must occur after the last definition to v on all possible execution paths, we define *No-More-Definitions* at node n ($NMD(n)$) to be the set of scalars that are not defined on any execution path from n to e. This function can be computed using dataflow analysis on the following equation:

$$NMD(n) = \begin{cases} \{V\} & \text{if } n = e \\ \bigcap_{s \in succ(n)} (NMD(s) - LocalDef(s)) & \text{otherwise} \end{cases} \quad (6.1)$$

For the example in Figure 6.3, the shaded boxes in Figure 6.3(c) indicate where $NMD(n)$ is true for the variable a. Note that there are two definitions of a, hence $\mathtt{a} \in NMD(n)$ for all nodes n dominated by these two nodes.



(a) Original code.

(b) Control flow graph.

(c) Inserting waits and signals.

Figure 6.3: Example of how waits and signals are inserted.

While it would be correct to insert `signal` instructions at all nodes n for which $NMD(n) \neq \{\}$, this may cause a single execution path from n to e to have many `signals`. We avoid redundant `signals` through the function $signal(n)$ which determines the placement of `signal` instructions:

$$signal(n) = \begin{cases} \{\} & \text{if } n = s \\ NMD(n) - \bigcap_{p \in pred(n)} NMD(p) & \text{otherwise} \end{cases} \tag{6.2}$$

Figure 6.3(c) shows the synchronization points for variables a and b for the original code in Figure 6.3(a).

6.3.3 Correctness Proof

Lemma 1 *On every execution path from s to e, there exists at least one* `signal` *for all* $v \in V$.

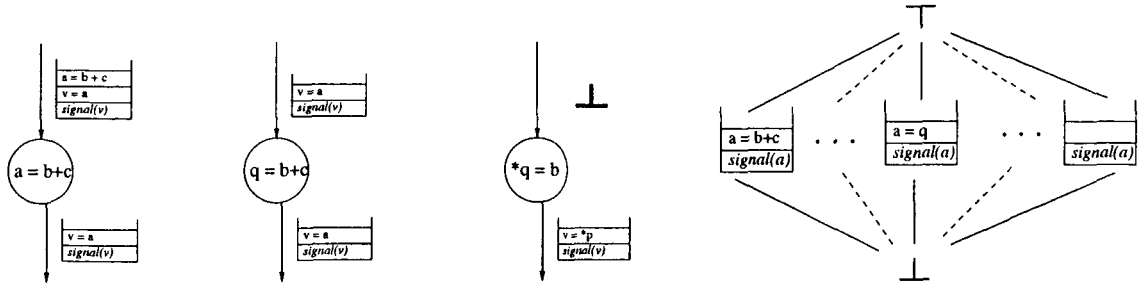
Proof: At the end node e, $NMD(e)$ contains v, by the definition of NMD. Since the variable v is defined in the epoch, v is not in $NMD(s)$, because the start node dominates all nodes in the graph. Since *signal* is true on any transition where a variable is in NMD to when it is not in NMD, and v leaves the set returned by NMD on the path from e to s, *signal* must be true on any path from e to s.

6.3.4 Performance Evaluation

The *U* bars in Figure 6.2 show the performance of the applications when synchronization is placed using the techniques described in this section, using the good set of regions. Each bar is normalized to the execution time of the original sequential version, such that bars less than 100 are speeding up. The *P* bar shows the impact of perfectly predicting all forwarded values—this shows an upper bound on the gains we can realize by optimizing synchronization.

Each bar in Figure 6.2 is broken down into four segments explaining what happened during all potential *graduation slots*. The number of graduation slots is the product of: (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors (4 in this case). The *fail* segment represents all slots wasted on failed thread-level speculation, and the remaining three segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *sync* portion represents slots spent waiting for synchronization for a forwarded location; and the *other* segment is all other slots where instructions cannot graduate.

As we see in Figure 6.2, synchronization is a significant bottleneck for most applications; hence placing the `signal` instructions as early as possible and the `wait` instructions as late as possible is not good enough—the critical forwarding path is still too large. There is much potential for improvement—as indicated by the *P* bars—by reducing the critical forwarding path. It is interesting to note that in half of the applications, eliminating the synchronization



(a) Dependence exists. (b) No dependence. (c) Unresolved dependence. (d) Lattice.

Figure 6.4: Illustration of the *transfer* function (parts (a), (b), and (c)) used for computing the value of *stack* in equation (6.3), and the lattice (part (d)) over which the *stack* dataflow analysis is defined.

bottleneck *increases* the amount of *failed speculation*, since the resulting increase in parallel overlap can lead to new occurrences of data dependence violations.

In many cases, the bulk of the computation in the critical forwarding path is not directly related to the forwarded scalar in question. If we can relocate this unrelated computation, the resulting smaller critical forwarding path will lead to improved performance. In the remainder of this chapter, we investigate the use of code motion to reduce the critical forwarding path.

6.4 Instruction Scheduling

The compiler can improve the performance of speculatively parallelized code by using scheduling techniques to move the `signal` operations (and the code that these operations depend upon) upwards through the control flow graph to reduce the length of the critical forwarding path and expose more parallelism. For example, closer examination of Figure 6.3 reveals that the forwarded value for variable a depends only on the result of a single addition. While the forwarding path between the `wait` and the `signal` shown in Figure 6.3 contains many instructions, only the following three instructions are really necessary to wait for, compute, and forward the new value of a:

```
wait(a);
a = a + 1;
signal(a);
```

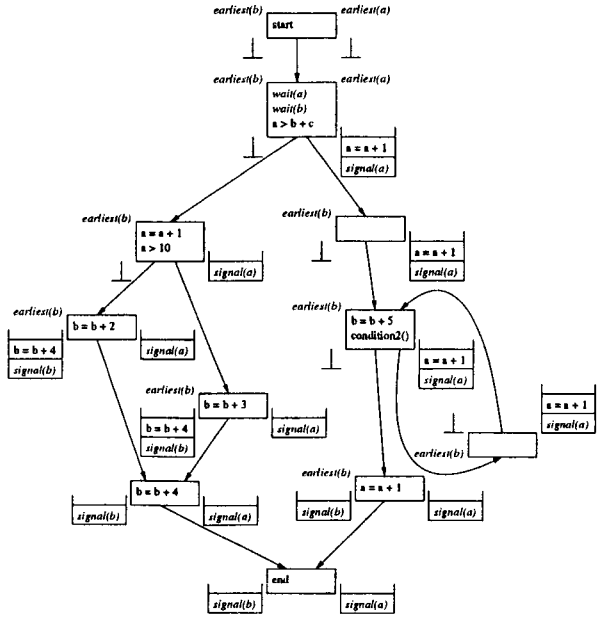
In the next section, we describe a scheduling algorithm that achieves this minimum critical forwarding path for this example.

6.4.1 Conservative Scheduling Algorithm

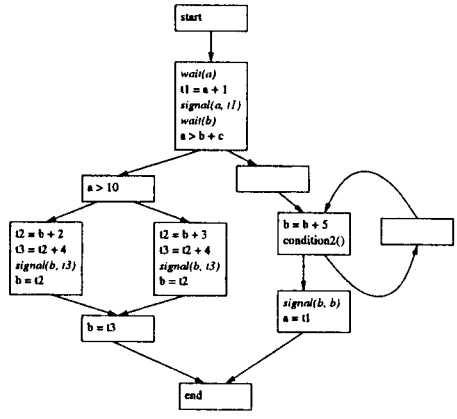
Similar to our algorithm for inserting synchronization, we also define our conservative scheduling algorithm over the set of communicating scalars V on the control flow graph $G = (N, E, s, e)$, with critical edges broken by synthetic nodes as described in the previous section. We initialize the algorithm by placing all `signals` at the exit node e. (It is equivalent to start all `signals` in the position indicated by our placement algorithm, but placing them at the end node simplifies the proof of correctness significantly.) Note that in our implementation of this algorithm, we have chosen only to move `signal` instructions (and the instructions they depend upon) upwards in the control flow graph; although the converse of this algorithm can be applied to moving `wait` instructions (and the instructions that depend upon them) downwards in the control flow graph, our experiments with moving waits downward showed little performance benefit since downward code motion is often blocked by data-dependent control dependences.

As we schedule the instructions, we must identify at each node the computation that the eventual `signal` depends upon. Since we cannot represent these computations as binary values, bit-vector analysis is inadequate. Hence at each node n, we keep a stack—denoted as $stack(n, v)$—of computation for each communicating scalar. This stack records the computations necessary to produce the value of a communicating scalar v if it is to be sent from the corresponding node.

The domain of the *stack* dataflow problem is the set of all possible configurations of the computation stack. This domain, along with the meet operator (described later), defines a semi-lattice. All nodes are initialized to $\top$. If a given



(a) Solutions to the *stack* and *earliest* problems.



(b) After code transformation.

Figure 6.5: Example of our conservative scheduling algorithm applied to the code in Figure 6.3.

node is found to be a safe place for the `signal` instruction then *stack* returns a non-empty stack of computations, otherwise *stack* returns $\perp$. The following dataflow equation computes $stack(n, v)$ at the exit of each node:

$$stack(n, v) = \begin{cases} \boxed{\texttt{signal } v} & \text{if } n = e \\ \prod_{m \in succ(n)} transfer(m, v, stack(m, v)) & \text{otherwise} \end{cases} \tag{6.3}$$

where the $transfer$ function is defined as follows:

- If $stack(m, v) = \top$, then $transfer = \top$.
- If the computation chain for v in the stack $stack(m, v)$ depends on a value w produced by node m, then the computation that produces w is added to the computation stack, as illustrated in Figure 6.4(a).
- If the computation chain in the stack $stack(m, v)$ does not depend on a value produced by the computation at node m, then $transfer = stack(m, v)$, as illustrated in Figure 6.4(b).

- If we cannot resolve the dependence between the computation chain for v and the computation in node m, we should stop the code motion; hence $transfer = \perp$, as illustrated in Figure 6.4(c).
- If a `wait` is issued for any exposed scalar in the computation chain, the code motion should stop; hence $transfer = \perp$.

The meet operator $\prod$ for the *stack* problem is defined over the set of all possible configurations of the computation stack. The lattice shown in Figure 6.4(d) defines the following operations for meet: if any input is $\perp$ then the output is $\perp$; if any input stack differs from any other input stack, then the output is $\perp$; otherwise, the meet operator returns the input stack, or $\top$ if all inputs are $\top$. The meet operator combined with the domain of the *stack* function defines a semi-lattice of height three, thus our dataflow problem is well-defined.

We also define the dataflow problem *earliest* to find the earliest synchronization point for each communicating scalar. *Earliest* is a bit-vector problem defined over the set of communicating scalars V on the control flow graph G. The $earliest(n, v)$ function is true at node n for v if no node prior to n is a safe place to schedule the `signal` on some execution path starting at s:

$$earliest(n, v) = \begin{cases} true & \text{if } n = s \\ \bigvee_{m \in pred(n)} (\neg safe(m, v) \wedge earliest(m, v)) & \text{otherwise} \end{cases} \tag{6.4}$$

where $safe(m, v) = (stack(m, v) \neq \perp)$, and all nodes are initialized to false.

Code Transformation: For each node that is both *safe* and *earliest* for a variable v, we insert the contents of v's stack either at the beginning of the node, or immediately after the computation that stopped code motion (a `wait` instruction or ambiguous pointer reference) if it exists. We replace references to v with temporary variables, and update the unscheduled computation to use these temporaries.

Figure 6.5(a) illustrates solutions for *stack* and *earliest* for the example shown earlier in Figure 6.3. *Earliest* is true for variable `a` only at the top node. The stack for the variable `a` at the top node contains only the two instructions required to compute `a`—this matches the optimal result we derived manually at the beginning of this section. Figure 6.5(b) shows the transformed program. Note that this transformation can either expand code size (by duplicating computations at join points), or reduce code size (by performing a form of common subexpression elimination at branch points). We observe in our experiments that the code segment size is increased by less than 1.3% for all benchmarks.

6.4.2 Correctness Proof

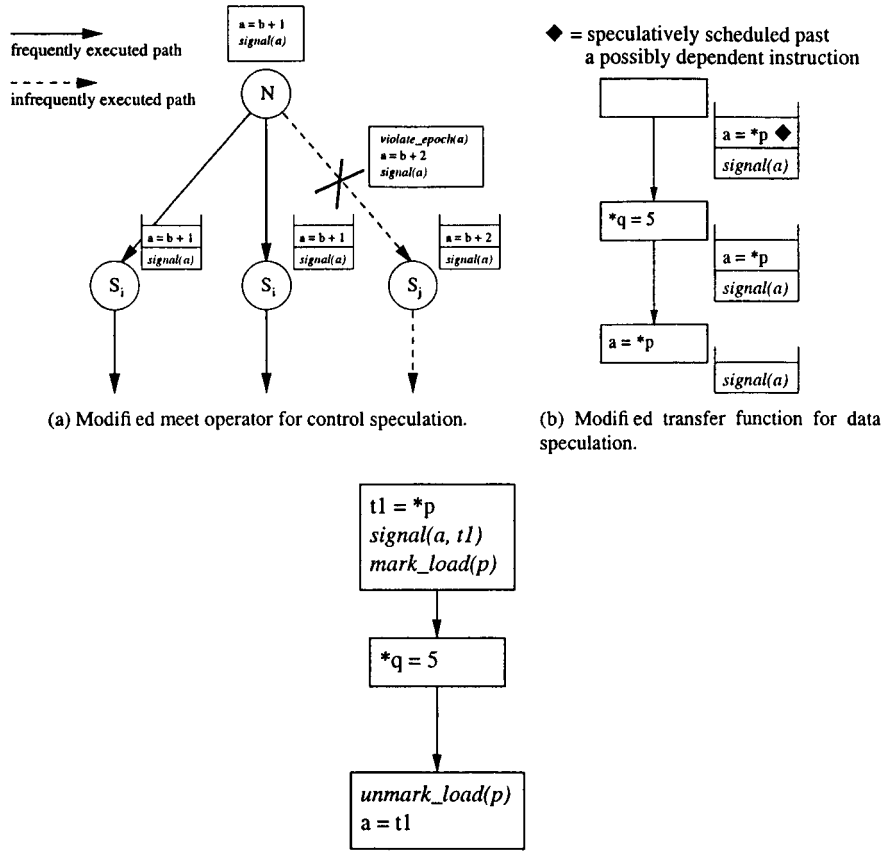
Lemma 2 *If node n is safe, then the computations in the computation stack delivers the correct forwarding value.*

Proof: By induction on the length of the execution path from (e, n).

Lemma 3 *On every execution path from the start node s to a node x where $v \in signal(x)$, there is one and only one node that is both safe and earliest for v, the communicating scalar.*

Proof: *Existence:* Node x is *safe* by the definition of *safe*, and node s is *earliest* by the definition of *earliest*. Assume that no node is both *safe* and *earliest* on the path $[s, x]$. We know that s is not *safe*, since s is *earliest*. Using the definition of *earliest*, we know that the successor must be *earliest*. However, by our assumption, the successor is not *safe*. By repeating this we can show that all nodes in the path are not *safe*—but we know that x is safe—hence our assumption is incorrect. Therefore there is at least one node that is both *safe* and *earliest* on the path $[s, x]$.

Uniqueness: Consider a path $[s, x]$. We want to show that only one node on this path is both *safe* and *earliest*. Assume that there are two unique nodes, n_1 and n_2, that are both *safe* and *earliest*. Since n_1 is *safe*, we know that all nodes m on any path between n_1 and e are *safe*, and for all m, $transfer(m, v, stack(m, v))$ is also safe. Since n_2 is *earliest*, the definition of *earliest* tells us that it has a predecessor q that is both $\neg safe$ and *earliest*. Hence q must not be between n_1 and n_2. By the definition of the meet function $\prod$ for *stack*, if q is $\neg safe$ it must have a successor that is $\neg safe$. Since we have eliminated critical edges in our graph, q can only have one successor—so q must be *safe*. Therefore q can not exist, which implies that n_2 can not be *earliest*. Hence there is no node n_2 that is both *safe* and *earliest* if there is already an n_1 on this path that is both *safe* and *earliest*.



(a) Modified meet operator for control speculation.

(b) Modified transfer function for data speculation.

(c) Code generation for data speculation.

Figure 6.6: Modified dataflow analysis for speculative instruction scheduling.

6.4.3 Aggressive Scheduling Algorithms

In the scheduling algorithm that we just described, the backward motion of `signal` operations (and the instructions on which they depend) can be obstructed for the following two reasons:

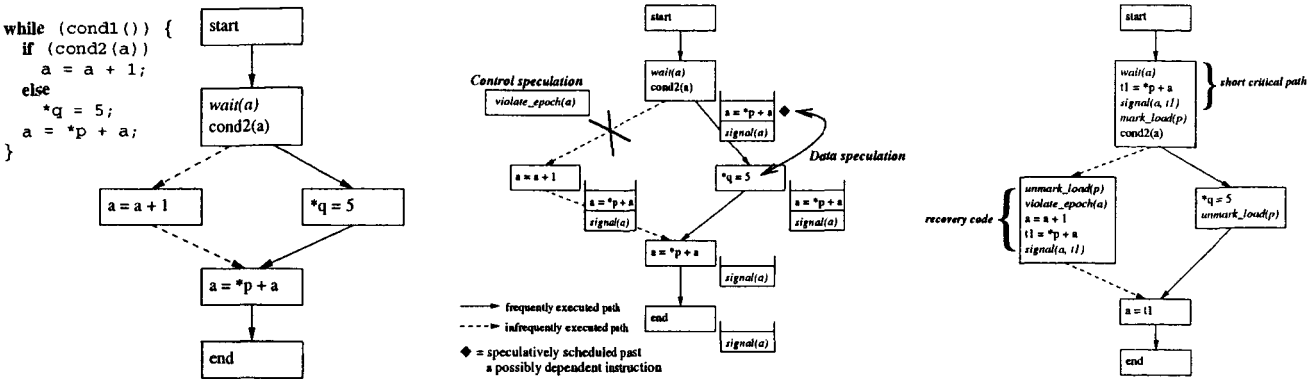
Control Dependences: If incompatible computation stacks from multiple execution paths meet at a single node during the backwards dataflow analysis, then code motion stops. This implies that our conservative scheduling algorithm cannot move instructions out of the `then` or `else` parts of an `if-then-else` statement unless those same instructions are executed along both conditional paths.

Data Dependences: A computation stack cannot be moved past a store instruction whose target address may alias locations referenced in the computation stack. This scenario often arises with writes through pointers or other forms of indirection.

Hence instructions are only scheduled at program points where the intra-epoch control and data dependences mentioned above have been resolved. Since both of these cases occur frequently in our programs, we would like to make scheduling more aggressive. In this section we will discuss both the compiler techniques and the hardware support necessary to allow for instruction scheduling beyond intra-epoch control and data dependences.

Scheduling Past Control Dependences

Dataflow analysis conservatively assumes that all execution paths are possible, and finds the minimal solution that satisfies all possible execution paths. In practice, however, only a small number of execution paths are frequently



(a) Original code. (b) Control flow graph. (c) Transformed code.

Figure 6.7: Illustration of how speculation on control and data dependences can be complementary.

executed at run-time. By taking this into account, we can schedule instructions aggressively for the common cases at the cost of possibly incurring an expensive recovery operation on the less frequently executed paths. Ball and Larus [6] proposed efficient methods to record execution paths that are taken by the program at run time, which allows us to identify the frequently executed paths.

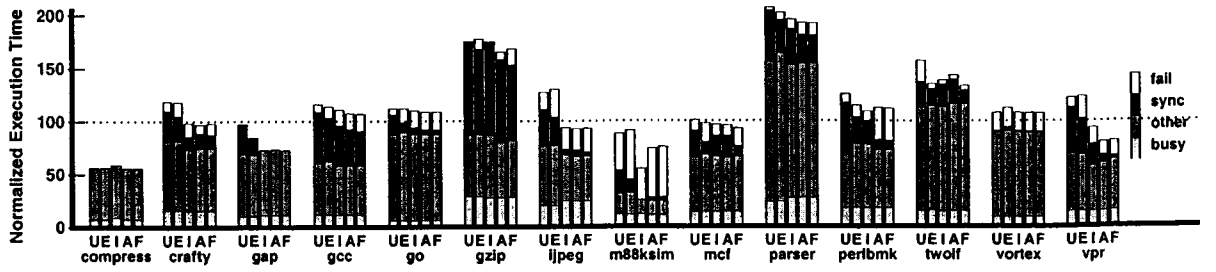
When we optimize for the common case, we will schedule code as early as possible, and `signal` the values as soon as they are available. If a less frequent path is taken, then this `signal` will have forwarded the wrong value to the next epoch—we need a mechanism to recover from this. For recovery, we first notify the next epoch that it received an invalid value, and then we forward the correct value to the next epoch. The notification of the next epoch is done using the `violate_epoch` instruction, which passes the identity of the communicated scalar—this instruction first discards the previously forwarded value, and then checks to see if the wrong value has already been consumed. If the incorrect value was consumed, then the epoch is violated and restarts; otherwise it is allowed to proceed. When the epoch reaches a `wait` instruction, it will stall until the correct value is received. If instructions are speculatively scheduled past branches (e.g., NULL pointer checks), then exceptions may occur in the scheduled code. When an exception occurs it should cause a violation, and a non-speculatively scheduled copy of the code should then be executed to ensure that the exception was real.

We have modified the scheduling algorithm from Section 6.4.1 to speculate on control dependences. We make the algorithm more aggressive by modifying the meet operator $\prod$ used in the *stack* dataflow analysis in equation (6.3), and add new nodes on infrequent edges which contain the recovery code. First, all possible execution paths through an epoch are enumerated, and a profiling run reports the number of times the epoch is executed as well as the number of times each execution path is taken. An execution path is considered *frequently executed* if the probability of taking this execution path when the epoch is executed is greater than a certain threshold.

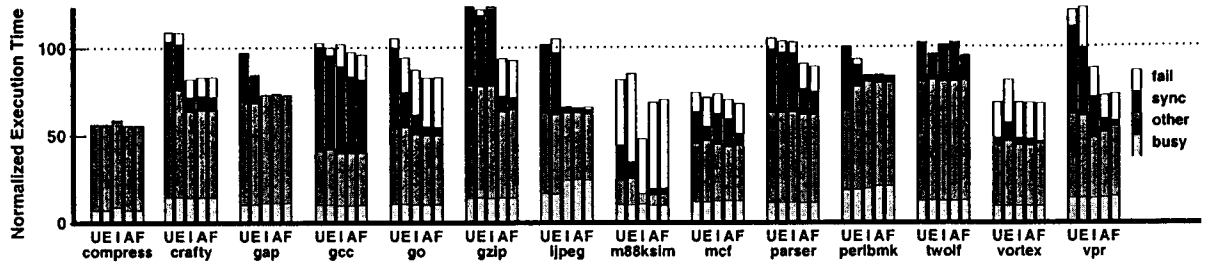
The meet operator $\prod$ for *stack* is modified as shown in Figure 6.6(a). When evaluating the meet operator $\prod$ at node n for the scalar v, we first operate on the set of successors, where the edge (n, s_i) is on a frequently executed path. Then for each node s_j, where (n, s_j) is not on any frequently executed path, we verify whether $transfer(s_j, v, stack(s_j, v))$ is compatible with the partially evaluated $stack(n, v)$. If this verification fails, then we add a new node on the edge (n, s_j) which contains a single `violate_epoch` instruction. We also make a minor change to the definition of *earliest* (shown earlier in equation (6.4)): it is always true for these new `violate_epoch` nodes, thereby making the scheduling algorithm automatically insert the `signal` stack at the appropriate point on the execution paths starting at the edge (n, s_j). Figure 6.6(a) illustrates how the two compatible computations on the frequently-executed nodes are scheduled above node N, while the infrequently-executed node on the right causes the next thread to be violated and re-executed with the correct value.

Scheduling Past Data Dependences

We now consider how our conservative scheduling algorithm can be extended to allow code motion beyond potential data dependences. Using the output from an automatic data dependence profiling tool, our compiler can reason about the likelihood of data dependence problems at run-time if the code associated with generating a particular `signal` operation is speculatively moved back ahead of a given potentially-conflicting store instruction. If a data dependence



(a) All regions.



(b) Good regions.

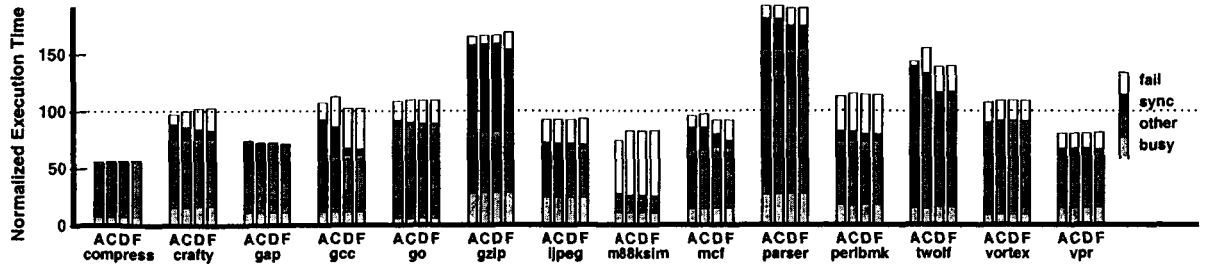
Figure 6.8: Impact of scheduling the critical forwarding path. For each benchmark, we show execution time for the speculatively parallelized regions of code normalized to that of the original sequential versions. *U* is unscheduled, *E* builds on *U* with hardware optimization of forwarded values, *I* schedules only loop induction variables, *A* schedules all forwarded variables, and *F* builds on *A* with hardware optimization of forwarded values.

does occur at run-time, we must first *detect* this situation, and then *recover* from our misspeculation. We *detect* data dependences by defining two new instructions: `mark_load` instructs the hardware to remember (i.e. "mark") the specified memory location. If any subsequent store modifies a marked location then the speculation fails. Once we have reached a point where the potential data dependence has been resolved then the `unmark_load` clears the mark on the memory location. If speculation fails or when an exception occurs, we *recover* by violating the current epoch—when the epoch restarts, it runs a different version of the code without speculative scheduling past data dependences. It is worth noting that this architectural support for speculative loads is quite similar to the `LD.A` and `CHK.A` instructions [35] available in the Intel IA-64 architecture. One important difference, however, is that when the speculative code motion fails in our case, the underlying TLS recovery mechanism rewinds execution to the start of the epoch; in contrast, under IA-64 the results of an `LD.A` instruction must be explicitly validated by a `CHK.A` instruction. (Further details on the implementation of `mark_load` and `unmark_load` can be found in our technical report [85].)

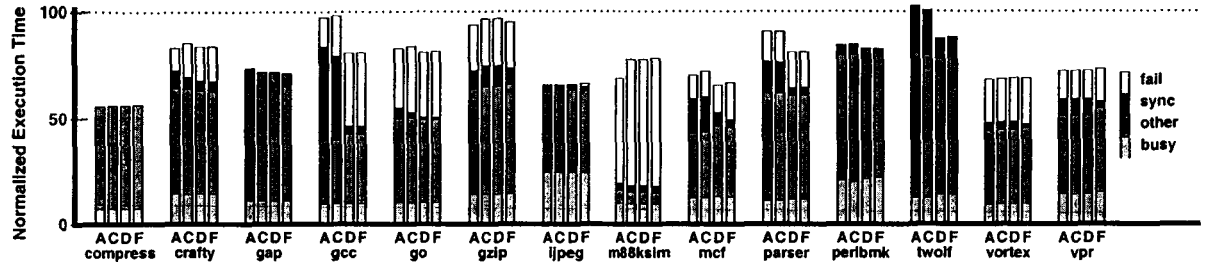
To implement scheduling across potential data dependences, we modify the *transfer* function described earlier in Section 6.4.1 (and used in equation (6.3)) as shown in Figure 6.6(b). When scheduling a stack of instructions across a potentially dependent store, we mark all potentially conflicting loads in the stack as being *possibly conflicting*. When two stacks are merged at node n through the meet operator $\prod$, any *possibly conflicting* marks are merged using logical *or*. At the time of code generation, we add a `mark_load` instruction after each *possibly conflicting* load. For all load instructions that are marked as *possibly conflicting*, an `unmark_load` is inserted at the original location of the load instruction.

Complementary Effects

Control and data dependence speculation can be complementary. Figure 6.7 shows an example where the combination of a control hazard and a data hazard prevent the code from being scheduled early, and where speculation on either type of hazard alone will not yield any benefit. By speculating on both control and data dependences in tandem, the computation of variable `a` can be moved upwards next to the `wait` operation, thereby resulting in a much shorter critical forwarding path for the common case.



(a) All regions.



(b) Good regions.

Figure 6.9: Impact of aggressive instruction scheduling and hardware optimization on region execution time. *A* is conservatively scheduled, *C* has aggressive instruction scheduling past control dependences, *D* has aggressive instruction scheduling past both control and data dependences, and *F* builds on D with hardware optimization of forwarded values.

6.5 Experimental Results

We now present our experimental results to quantify the performance impact of the scheduling algorithms described in the previous section. We include a comparison with hardware-based techniques that are also designed to reduce synchronization stalls under TLS [69], as well as a comparison between our conservative algorithm and the Multiscalar scheduling algorithm [77].

6.5.1 Impact of Conservative Scheduling

Figure 6.8 shows the impact of our conservative scheduling algorithm on parallelized region performance. (Recall the definitions of *"all regions"* and *"good regions"* presented earlier in Section 6.2.2.) Note that in most cases, the unscheduled version (*U*) slows down relative to the original sequential version (i.e. the height of the bar is greater than 100).

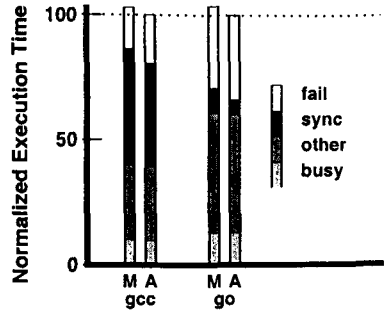
We first evaluate the performance of a hardware technique (*E*) that we described in an earlier publication [69] to schedule the critical forwarding path and predict forwarded values for the sake of eliminating synchronization stalls. We observe significant improvements for GAP, PERLBMK, and TWOLF, as well as for GO when poor regions are pruned (i.e. *good regions* in Figure 6.8(b)).

When our scheduling algorithm is applied to loop induction variables alone (*I*), the synchronization stall times decrease significantly (more than 50% for 9 of the 14 applications), and many applications now enjoy significant speedups. On average, *all regions* are improved by 11.8% and the *good regions* by 13.6%. Only COMPRESS performs slightly worse under this optimization. We note that reduction of synchronization stall time does not always translate directly into improved performance. For example, synchronization time is greatly reduced for GO when loop induction variables are scheduled (*I*), but the resulting increased parallel overlap exposes more data dependences across threads, and failed speculation becomes a new bottleneck that offsets much of the potential performance gain.

When all forwarded variables are scheduled (*A*), we see in Figure 6.8 that many applications enjoy additional improvement, while the performance of three applications is degraded. The worst degradation occurs in M88KSIM for the same reason mentioned above: the reduced critical forwarding path exposes inter-thread memory dependences that

```
for (insn = target; insn; insn = next) {
  rtx this_jump_insn = insn;
  next = NEXT_INSN(insn);
  switch (GET_CODE(insn)){
    case ... : ...
    case INSN:
      if (GET_CODE(PATTERN(insn)) == USE){
        ... ; continue;
      } else if (GET_CODE(PATTERN(insn)) == CLOBBER){
        continue;
      } else if (GET_CODE(PATTERN(insn)) == SEQUENCE){
        for (i=0; i<XVECLEN(PATTERN(insn),0); i++){
          ...
        }
      }
    }
}
```

(a) Simplified loop from GCC (`reorg.c:2680`).



(b) Performance comparison.

Figure 6.10: Comparison with the Multiscalar scheduling algorithm. *M* approximates the Multiscalar scheduling technique, and *A* is our conservative scheduling algorithm. Execution time of *good regions* in part (b) is normalized to *A*.

previously were synchronized indirectly, thereby resulting in a large increase in failed speculation for this particular case. This example illustrates the complex interactions that can occur among different potential bottlenecks under TLS, and suggests that there is still room for improvement in future research by attacking these other bottlenecks.

We now consider the effectiveness of the compiler versus the hardware at optimizing the critical forwarding path. Our first observation from Figure 6.8 is that our conservative scheduling algorithms (*I* and *A*) outperform the hardware-only technique (*E*) in nearly every case. To evaluate whether the compiler and the hardware are complementary, we supplemented the compiler's efforts with hardware support (*F*) for optimizing forwarded values (as was done for (*E*)). These hardware mechanisms offer a slight additional benefit for only a few cases, suggesting that hardware mechanisms for optimizing the critical forwarding path are largely unnecessary given proper compiler support.

In summary, we observe that code motion, even if it is conservative, is an effective way to reduce the critical forwarding path, and that the compiler appears to be better suited to this optimization than hardware. While most applications in Figure 6.8 have enjoyed substantial reductions in synchronization stall times (*sync*), there are still a handful of cases where this bottleneck remains significant. We now investigate whether our more aggressive scheduling algorithms (based on control and data dependence speculation) can reduce these stall times further.

6.5.2 Impact of Aggressive Scheduling

Our control and data dependence speculation algorithms exploit path frequency information and data dependence information gathered from a profile of each application. For control speculation, we consider any path through an epoch that was executed at least 5% of the time to be "*frequently executed*" (as discussed earlier in Section 4.2.1). For data dependence speculation, we speculatively move code back across stores or function calls unless there is more than a 15% chance of this resulting in a data dependence violation. Although experimentation with these threshold values showed that the best values vary between applications, we chose to use these fixed values throughout this chapter.

Figure 6.9 shows the impact of aggressive instruction scheduling and hardware optimization on region execution time. The first bar (*A*) for each application shows the performance of conservative scheduling (as seen earlier in Figure 6.8); the *sync* portion of these bars shows the potential gain from better scheduling, with GCC, MCF, PARSER and TWOLF being the most significant. The second bar (*C*) shows the performance of speculatively scheduling past control dependences alone. We observe that performance improves or degrades slightly for several applications, except for M88KSIM and TWOLF (*all regions*) for which the degradation is more severe: these contain cases where the code has not moved a significant distance, and where the benefits of the code motion are not sufficient to overcome the costs of misspeculation. Our scheduling algorithm could make more informed tradeoffs if it took this code motion distance into account, rather than always moving code whenever possible (as it does now).

For the *good regions* cases in Figure 6.8(b), speculatively scheduling past both control and data dependences (*D*) decreases synchronization time by an average of 19.8%. Speculative scheduling results in a 0.3% average performance degradation for the *all regions* cases, and a 5.8% average performance improvement for the *good regions* of

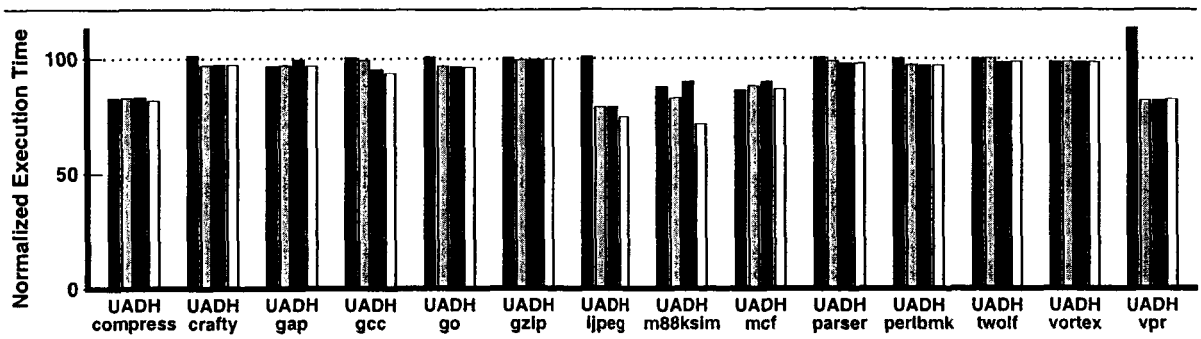


Figure 6.11: Impact on program execution time of aggressive instruction scheduling and hardware optimization when only the good regions are speculatively parallelized. *U* is unscheduled, *A* is conservatively scheduled, *D* has aggressive instruction scheduling past both control and data dependences, *H* builds on *D* with hardware optimization of memory and forwarded values.

applications for which more than 6% of execution time is spent on synchronization—most notably, GCC, PARSER, and TWOLF, which speed up by 17.1%, 19.1%, and 14.8% respectively. These observations suggest that the overheads of speculative scheduling are too large to apply this technique liberally, but that it can be quite effective for certain regions of code (where synchronization stalls remained a problem after conservative scheduling) if applied selectively.

Finally, we again supplement the compiler's efforts with hardware support (*F*) that schedules the critical forwarding path and predicts forwarded values to eliminate synchronization [69]. We observe that improvements from this additional hardware support are negligible, and that such hardware support is not necessary with sufficiently aggressive compiler optimization.

6.5.3 Comparison of Conservative Scheduling with the Multiscalar Algorithm

Since the Multiscalar scheduler [77] is essentially a dataflow algorithm that only traverses the control flow diagram once, we can estimate its operation by constraining our conservative scheduling algorithm: we modify the meet operator such that it returns $\perp$ whenever $\top$ meets with any value that is not $\top$—this way, the modified dataflow analysis will converge during the first iteration.

Figure 6.10(a) shows a simplified version of a loop in GCC (at line `reorg.c:2680`) that highlights the advantage of the more general dataflow approach of our conservative scheduling algorithm over the Multiscalar algorithm [77]. While the original version of this loop has multiple variables that are forwarded, we focus on the variable `insn`. The Multiscalar scheduler cannot move the update and forward of `insn` above the inner loop in the `case` statement, while our approach iterates to a dataflow solution where it can.

Figure 6.10(b) shows a performance comparison of our conservative scheduling technique with that of the Multiscalar algorithm for the *good regions* cases of the two applications where there was a significant difference in performance: i.e. GCC and GO. Compared with the Multiscalar algorithm, our conservative scheduling approach reduces synchronization time by 10% for GCC and by 39% for GO, which in turn reduces the respective region execution times by 3.0% and 3.7% relative to the Multiscalar approach. Again, this result is not surprising since the Multiscalar algorithm was designed for smaller, simpler regions.

6.5.4 Impact on Overall Program Performance

The goal of TLS and our techniques for improving its efficiency is to exploit chip multiprocessors and other multithreaded machines to improve the performance of *programs*. Up to this point, we have evaluated the performance of our optimizations on the *regions* of each program that have been speculatively parallelized. Figure 6.11 shows the impact on program execution time of TLS with compiler and hardware optimization when only the *good regions* are speculatively parallelized. The first bar (*U*) shows the unscheduled version, the second (*A*) shows conservative scheduling. We observe that IJPEG and VPR enjoy a tremendous benefit from conservative scheduling while CRAFTY, GO, and M88KSIM show more modest improvements. The third bar (*D*) shows aggressive scheduling past control and data dependences, which shows a significant improvement for GCC but degrades performance for several other

applications, indicating that we must be more selective when applying this technique. The fourth bar (*H*) shows the additional impact of hardware optimization of both forwarded values and memory values [3, 20, 48, 69], which improves GCC, COMPRESS, IJPEG, and M88KSIM significantly. For the remaining applications, our hardware techniques cannot significantly improve upon the the performance of the compiler. (Note that our hardware technique for automatically synchronizing data dependences [69] is not the most aggressive approach possible [55].) With our most aggressive compiler and hardware techniques, we have improved program performance of 6 of 14 applications by 6.2–28.5%, and we improved half of the others by 2.7–3.6%.

6.6 Summary

The critical forwarding path is an important bottleneck to overcome when trying to extract parallelism from many important programs using TLS. In this chapter, we have proposed and evaluated a range of scheduling algorithms that the compiler can use to reduce the impact of the critical forwarding path. Loop induction variables were the largest performance bottleneck, and our conservative scheduling technique effectively eliminates their impact on performance. By applying conservative scheduling to all synchronized variables, we saw additional performance gains. These results demonstrate that the compiler can be effective in reducing the performance impact of the critical forwarding path without requiring any additional hardware support beyond what is normally needed for TLS.

To further reduce the critical forwarding path for the handful of applications where synchronization stalls were still a concern, we proposed and evaluated scheduling techniques based upon speculative code motion that require some additional hardware support to preserve correctness. We found that scheduling speculatively past control dependences alone offered little performance benefit. However, scheduling speculatively past both control and data dependences resulted in substantial performance gains for a number of applications. In particular, GCC—which is the most challenging application among the set that we considered—was the largest benefactor from this speculative scheduling. GCC also highlighted the performance advantages of our robust dataflow-based approach to scheduling compared with the previous state-of-the-art technique.

The bottom line from this study is that the critical forwarding path bottleneck for TLS is best addressed by the compiler rather than through elaborate hardware mechanisms. If hardware resources are to be devoted to this problem, they are best spent on implementing the instructions necessary to support speculative scheduling of `signal` operations (and the instructions they depend upon) past both control and data dependences.

Chapter 7

Compiler Optimization of Memory-Resident Value Communication

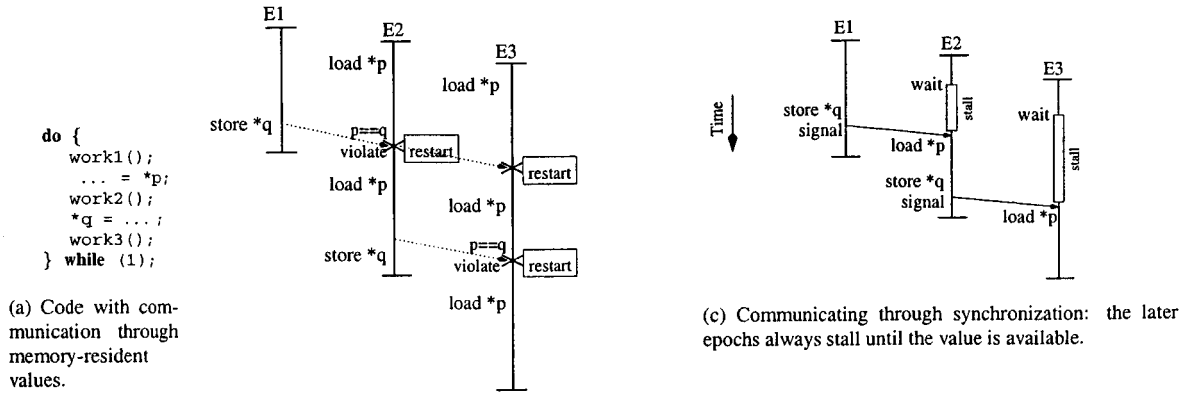
7.1 Introduction

In thread-level speculation, since speculation failure incurs a high cost it should only be invoked occasionally—we must seek alternative methods to deal with frequently occurring data dependences between speculative threads. One way to avoid speculation failures caused by data dependence violations is to synchronize frequently-occurring data dependences, as explained in the previous chapter. Figure 7.1 shows a loop example that the compiler has speculatively parallelized by turning each loop iteration into an epoch. In each epoch a value is loaded through the pointer p and another value is stored through the pointer q. When p in a later epoch points to the same memory location as q in an earlier epoch, there is a *read-after-write* dependence. Figure 7.1(b) and 7.1(c) show two methods to communicate a value between the two epochs to satisfy this dependence. The first method is *speculation*: the consumer epoch executes assuming there is no data dependence with previous threads and is re-executed if the hardware detects a dependence violation. The second method is *synchronization*: the consumer epoch stalls and waits for the producer epoch to produce and forward the correct value. Synchronization serializes parallel execution and only allows partial overlap between parallel epochs, but is more efficient than speculation when data dependences occur frequently since restarts are avoided. [1]

The existence of aliases between memory accesses makes it more difficult to synchronize accesses to memory-resident values than accesses to scalar values. Previous work on compiler optimization for inter-epoch value communication [84] focuses on communicating register-resident *scalar* values. It shows that: (i) compiler-inserted synchronization and forwarding can communicate scalar values efficiently between epochs; and (ii) instruction scheduling techniques are essential for reducing the *critical forwarding path* created by such synchronization. However, these techniques cannot be directly applied to communicate memory-resident values since the compiler is unable to identify the producer and the consumer of a data dependence statically. Figure 7.3(a) shows three epochs running speculatively in parallel. *Load *p* can potentially depend on any of the five stores in the figure, although each access to the memory uses a different pointer. The compiler must prove that *load *p* depends on *store *q* in all possible executions before synchronizing the two instructions and directly forwarding a value between them. Such a proof is difficult and sometimes impossible to construct. If the compiler decides to synchronize *store *q* and *load *p* without such a proof, we must confirm at runtime that (i) p and q refer to the same memory location, and that (ii) stores through pointers y and z do not modify this location.

Previously, a number of studies [21, 55, 68] propose using hardware implementations to dynamically insert synchronization for frequently occurring and unpredictable data dependences in TLS. Moshovos *et. al.* [55] demonstrated how to identify frequently occurring data dependences with a centralized structure. However, a centralized structure can limit performance [38] and is difficult to scale. On the other hand, the distributed version of this scheme is complex since it involves replicating the tables which predict/synchronize load-store pairs and keeping them coherent via broadcast. In a distributed environment, it is relatively easy for the hardware to dynamically identify loads that frequently cause speculation to fail using hardware lookup tables, but more involved for the hardware to identify the corresponding stores. For the hardware to dynamically identify an inter-epoch dependence pair it has to (i) compare

[1] Results shown in this section can also be found in our previous publications. [86]



(a) Code with communication through memory-resident values.

(b) Communicating through speculation: if p and q alias between epochs then the later epochs are violated and restarted with the correct value.

(c) Communicating through synchronization: the later epochs always stall until the value is available.

Figure 7.1: Performance trade-off of using speculation versus synchronization under TLS.

the addresses accessed by loads and stores in different epochs, and (ii) dynamically determine whether a store is the last store that modifies an address in an epoch. To avoid this complexity, recent proposals for hardware-inserted synchronization [21, 68] choose to delay the load instructions until previous epochs have committed. However, such simplification tends to over-synchronize parallel execution, trading time spent on failed speculation for time spent on synchronization. In contrast, compilers have the advantage of knowing the entire program, thus can determine which stores are more likely to produce the desired value and, therefore, only stall the consumer until the value is produced (rather than waiting for the entire producer epoch to complete). Compilers can also schedule instructions to produce the forwarded value early to reduce synchronization time. Furthermore, compiler-inserted synchronization avoids hardware complexity by eliminating lookup tables used by hardware proposals to identify frequently occurring loads.

7.1.1 Our Approach: Compiler-Inserted Synchronization for Memory-Resident Values

In this chapter, we propose to use the compiler to insert explicit synchronization to communicate values more efficiently for inter-epoch data dependences that occur frequently. In our approach, we first identify frequently occurring data dependences using profiling information, then insert *signal* and *wait* instruction pairs, the same synchronization primitive as sued for synchronizing communicating scalars [84], to create point-to-point synchronization and to forward the values involved in the dependences. We also describe the hardware support required to verify that the synchronized load and store are indeed dependent at runtime and to guarantee recovery from incorrect execution if they are not. Details of this hardware support are in Section 7.2.2.

The compiler decides where to insert synchronization based on the output of a software-only instrumentation-based tool. In our experiment this tool records all accesses to the memory and matches all dependent load and store instructions. Pointer analysis [53, 80], especially probabilistic, inter-procedural and context-sensitive pointer analysis [8, 15, 46] could help us obtain this information with less detailed profiling information. Data dependence profiling and compiler insertion of synchronization are described in more detail in Section 7.2.3.

7.1.2 Performance Impact of Failed Speculation

To estimate the performance potential of compiler-inserted synchronization for memory-resident values, we study TLS execution with optimal memory-resident value communication. Figure 7.2 shows the potential impact of reducing failed speculations in the parallelized regions of a program on a four-processor chip multiprocessor that supports TLS (detailed in Section 7.3). Each bar in Figure 7.2 is broken down into four segments explaining what happens during all potential *graduation slots*. The number of graduation slots is the product of: (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors (4 in this case). The *fail* segment represents all slots wasted on failed thread-level speculation, and the remaining three segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *sync* portion represents slots spent waiting

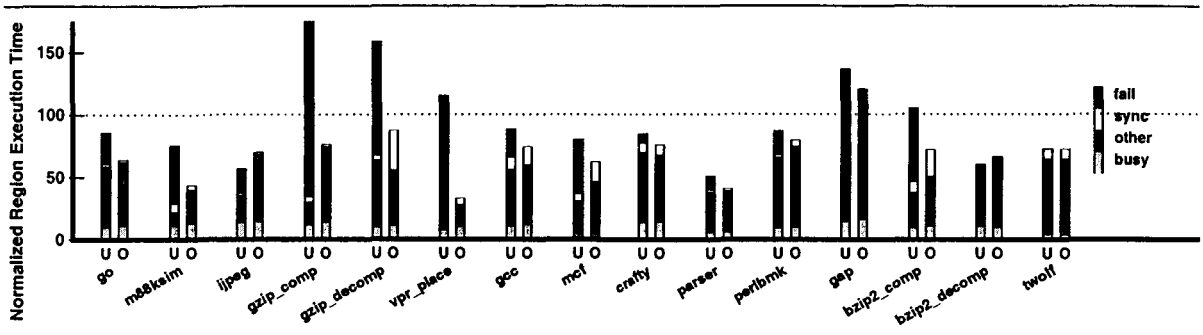


Figure 7.2: Potential impact of reducing failed speculation. For each benchmark we show execution time on four processors (for the speculatively parallelized regions of code) normalized to that of the original sequential version. **U** is unoptimized, without synchronization of memory resident values; **O** shows the impact of perfect (oracle) prediction of loads of memory resident values.

for synchronization for scalar values (scalar values are communicated using explicit synchronization); and the *other* segment is all other slots where instructions cannot graduate. The **U** bars represent the execution time of the benchmark when run in parallel using TLS. Each bar is normalized to the execution time of the original sequential version, and hence bars less than 100 are speeding up. The best we can possibly do to reduce speculation failure is to prevent any data dependence speculation from failing. We measure this ideal behavior by running the same benchmarks with a hypothetical model that perfectly forwards the values needed by all load instructions such that no failed speculation nor synchronization stall ever occur due to accesses to the memory (**O** bars). We find that for most benchmarks, eliminating failed speculation results in a substantial performance gain.

7.1.3 Related Work

Previous work on synchronizing loop-carried data dependences for DOACROSS loops [7, 14, 27, 62, 81] only focuses on array-based numeric codes. Our technique applies to arbitrary control flow and memory access patterns in general-purpose programs, and is able to (i) forward data for dependences that may or may not occur, and (ii) ensure correct execution if subsequent stores invalidate the data that have already been forwarded. Prior to our work, Sura *et. al.* [70] used the compiler to insert memory fence instructions to map the consistency model at the programming language level to the consistency model offered by the hardware. Correct execution must be ensured through this mapping, hence, their compiler analyses are conservative. In our case, correctness is ensured by the underlying hardware and we synchronize frequently occurring data dependences strictly to improve performance, hence, the compiler analyses used in this work to insert synchronization can be more aggressive.

To avoid excessive failed speculation when using TLS, two types of hardware mechanisms: *value prediction* [21, 51, 55, 59, 63, 68] and *synchronization* [21, 55, 68] have been proposed. Value prediction allows the consumer of a potential data dependence to use a predicted value, avoiding a dependence violation if the prediction is correct. Hardware support for automatic synchronization identifies store-load dependences that frequently cause violations and attempts to synchronize them dynamically. The various implementations of these two hardware mechanisms are discussed below.

Dynamic synchronization of memory accesses can benefit both uniprocessors and multiprocessors. In superscalars, loads are usually issued as early as possible, but no earlier than prior stores that write to the same memory address to avoid memory-order violations. Chrysos and Emer [18] present a design that uses a prediction table for synchronizing dependent store-load pairs in an out-of-order issue uniprocessor. Moshovos *et. al.* [55] demonstrate how to implement a hardware-based synchronization mechanism in the context of a Multiscalar processor (a thread-speculative chip-multiprocessor) using centralized lookup tables to match dependent load/store pairs from different processing units.

A major drawback of previous proposals is the need for centralized lookup tables which can limit performance and are difficult to scale. Two groups [21, 68] propose alternative implementations to manage synchronization information in a distributed manner. Cintra and Torrellas [21] propose building a distributed hardware lookup table to keep track of frequently occurring violations. They divide data dependences into three categories and handle them accordingly. For violations caused by false dependences, they optimistically allow the consumer to proceed and use the per-word

access bits in its cache hierarchy to check for correctness before committing. In the case of a true dependence where the value is predictable, the consumer uses a predicted value and later verifies the value before committing. In the case of a true dependence with an unpredictable value, violations are avoided by stalling the consumer until the producer has committed. Their evaluation shows that these optimizations can substantially improve value communication for floating point benchmarks.

In prior work we evaluated the use of value prediction to communicate predictable values and the use dynamically inserted synchronization to communicate unpredictable values [68]. We found that value prediction and dynamic synchronization can incur a significant cost and should only be applied to those dependences that limit performance. Loads that frequently cause violations are delayed until the producer epoch commits rather than until the desired value is produced, due to the difficulty in identifying dependent store-load *pairs*. Thus, this dynamically inserted synchronization tends to serialize parallel execution more than necessary.

In another prior work we explored the compiler's ability to improve scalar value communication, and showed that compilers can communicate scalar values efficiently between epochs [84]. By targeting scalar values, we have been able to use traditional data-flow analysis to find all reads/writes to the same data item and identify the producer and the consumer of a data dependence. We conclude that the key to efficiently communicating scalar values between epochs is to reduce the *critical forwarding path* created by synchronization, a task effectively accomplished by the compiler through instruction scheduling. Although this chapter focuses more on reducing the cost of violations instead of reducing the impact of the synchronization we insert to avoid violations, we attempt to evaluate the significance of reducing the cost synchronization for communicating memory-resident values in Section 7.4 through idealized experiments.

7.1.4 Contributions

In the context of thread-level speculation, this chapter makes the following three contributions. First, this is the first attempt to explore a compiler-based approach to improving the communication of memory-resident values. We demonstrate the automatic insertion of synchronization and forwarding primitives, and also how to ensure correct execution when forwarding potentially aliased values. Second, we show that compiler-inserted synchronization can reduce the amount of failed speculation caused by frequently-occurring dependences, and hence improve performance significantly for some applications. Finally, we compare and contrast our approach for compiler-inserted synchronization of memory resident values with a recently proposed hardware technique [68] and demonstrate that the hardware and compiler can work in tandem.

7.2 Synchronizing Memory-Resident Dependences

Previous research [84] has shown that compiler-inserted synchronization can effectively communicate scalar values between epochs and improve program performance by boosting the efficiency of parallel execution. In this chapter, we extend this work to communicate memory-resident values. Synchronizing frequently-occurring memory-resident values is, however, more complicated due to the existence of potential aliasing (i.e., a pointer through which the memory location in question is unexpectedly modified). In this section, we first describe how the compiler identifies and synchronizes register-resident scalar values, then point out the differences between communicating register-resident values and memory-resident values. We also describe how the compiler can explicitly synchronize accesses to memory-resident values and avoid failed speculation.

7.2.1 Synchronizing Register-Resident Values

We can identify scalars that require synchronization using traditional data flow analysis. Scalar synchronization [84] is applied to the set of local *communicating scalars* (i.e. those defined in the scope of the enclosing procedure), which we define as any scalar which is *live* between epochs and does not have its address taken. Since each communicating scalar is allocated to a register (assuming it is not spilled), we also refer to the values they hold as *register-resident* values. For each communicating scalar, the compiler inserts `wait` and `signal` instructions to synchronize and forward the value. The `wait` instruction stalls until a value is produced and forwarded by the producer epoch through a `signal` instruction.

The following characteristics of register-resident values make them easier to synchronize than memory-resident values: (i) there is no aliasing in accessing scalar values, all accesses (reads or writes) must explicitly refer to the

single register name; and (ii) static instructions that access communicating scalars only occur in the loop body being optimized, not in the procedures called from the loop body. Thus, it is relatively straightforward to identify all accesses to these values and to use data-flow techniques to determine the last definitions and the first exposed uses within an epoch.

7.2.2 Synchronizing Memory-Resident Values

Unfortunately, the mechanism for forwarding register-resident scalar values with `signal` and `wait` instructions cannot be directly applied to forwarding memory-resident values for two reasons. First, we are unable to decide whether two memory accesses refer to the same data item using traditional data-flow analysis when the same location can be accessed using different names through pointers. Second, the existence of potential aliasing in accessing memory-resident values make it difficult and sometimes even impossible to determine the last definition and the first exposed use of a data item within an epoch. Thus, as opposed to only synchronizing frequent dependences at definite program points, we now synchronize *probable* data dependences for memory-resident values.

Now we take a close look at how aliasing in memory accesses makes our problem more difficult. An inter-epoch dependence occurs between a store and a load if: (i) the store occurs in a logically earlier epoch, (ii) both the store and the load access the same memory address, and (iii) no other store, between the store and load in question, modifies this address. Figure 7.3(a) shows three epochs executing speculatively in parallel. Assume that *load *p* could depend on any of the five store instructions while, however, it depends on *store *q* most frequently, thus, we want to synchronize and forward a value between this pair. Traditional pointer analysis [8, 15, 80] may help us reduce the set of pointers that p aliases to, but could not provide the set of *frequently* dependent instructions that we need. For instance, *must*-alias pointer analysis could not identify *likely* dependences, such as *load *p* and *store *q*, hence would not synchronize them. On the other hand, *may*-alias pointer analysis would indicate that *load *p* may depend on any of the five store instructions, hence they should all be synchronized. Since neither provides us with the desired information in this situation, we need profiling-based tools that identify *likely* dependences [16, 15]. We also need mechanisms that allows us to synchronize these likely data dependences, and to ensure correct execution for whatever dependences actually occur at runtime.

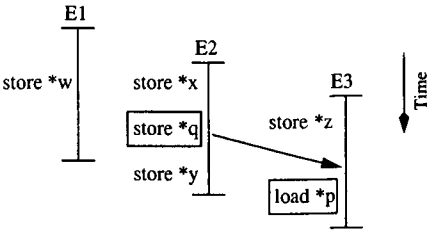
In the rest of this section we describe the hardware mechanism for synchronizing memory-resident values while preserving correct execution, using Figure 7.3(b) as our guide.

The producer of the forwarded value still has to store the value to memory, since it may still be read from memory by other parts of the program. The producer also has to communicate the forwarded value and its address, through the *signal* instructions. In addition, the producer has to be able to detect if the wrong value was forwarded—this is done by storing the address in the *signal_address_buffer*, a small per-cpu buffer which is used to make sure that no later store in the epoch writes to the same memory location.

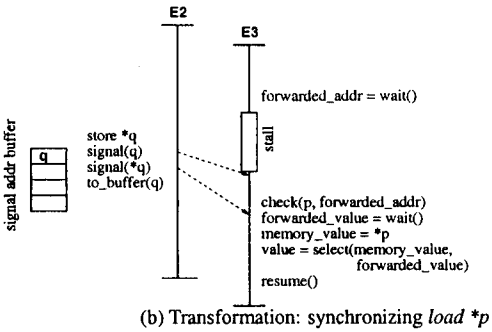
The consumer of the forwarded value first has to wait for the value and its address to arrive, through the *wait* instructions. The consumer then checks to see if the addresses match (to make sure a useful value was received), and if so sets a cpu-local flag called *use_forwarded_value*. The consumer then issues a load to the speculative cache. If the *use_forwarded_value* flag is set when this load is issued, this instruction only accesses the speculative cache and will *not* cause a violation. This load also checks to see if the value has been overwritten locally and clears the *use_forwarded_value* flag if it is. The value of the *use_forwarded_value* flag then determines whether the forwarded value or the value loaded from memory is used in subsequent computation, and when we are finished the *use_forwarded_value* flag is reset.

We now describe how correctness is ensured by describing all possible data dependences that may occur. When a true data dependence occurs between *store *q* and *load *p*, the forwarding mechanism forwards the correct address and value, then the forwarded value is used. If *load *p* depends on *store *w* or *store *x*, the forwarded address q cannot point to the same location as p. Thus, the *use_forwarded_value* flag is not set, the *select* instruction will choose *memory_value*, and the underlying hardware that supports TLS will ensure correct execution. If p, q and y all point to the same memory location, the forwarding instruction will forward the correct address, but a wrong value. The producer epoch will notice that it is storing to an address that is already in the *signal_address_buffer*, and send a signal which restarts the consumer epoch. If *load *p* depends on *store *z*, *use_forwarded_value* flag is reset by the load instruction and we will use the value loaded from the memory. This is correct, since this memory access is not exposed and the local cache holds the correct value.

It is possible that on some paths through an epoch the value is never produced. In this case, the producer epoch should still signal the consumer epoch by sending a `NULL` value in the address field, so that the consumer does not wait indefinitely. If p points to a valid address then it will not match this `NULL` pointer, and the load in the consumer



(a) Original program: *load *p* often depends on *store *q*.



(b) Transformation: synchronizing *load *p* and *store *q*.

Synchronization operation	Description
store *q;	The original store operation.
signal(q);	Forward the address q to the next epoch.
signal(*q);	Forward the value stored as well.
to_buffer(q);	Save the address q in the signal address buffer.
forwarded_addr = wait();	Wait for the address to arrive from the previous epoch.
check(p, forwarded_addr);	If *p* equals *forwarded_addr* set the *use_forwarded_value* flag. Loads issued while this flag is set will not cause violations.
forwarded_value = wait();	Wait for the value to arrive from the previous epoch.
memory_value = *p;	Load a value from the memory system using the load operation. If the address *p* has been previously modified by the current epoch, this instruction clears the *use_forwarded_value* flag. If the *use_forwarded_value* flag is set when this load is issued, this instruction only accesses the speculative cache and will *not* cause a violation.
value = select (memory_value, forwarded_value);	If *use_forwarded_value* flag is set, select *forwarded_value*, otherwise, select *memory_value*. The selected value is placed in *value*.
resume();	Reset the *use_forwarded_value* flag.

(c) Description of operations inserted for synchronization.

Figure 7.3: Program transformation to synchronize frequently occurring memory-resident dependences between epochs.

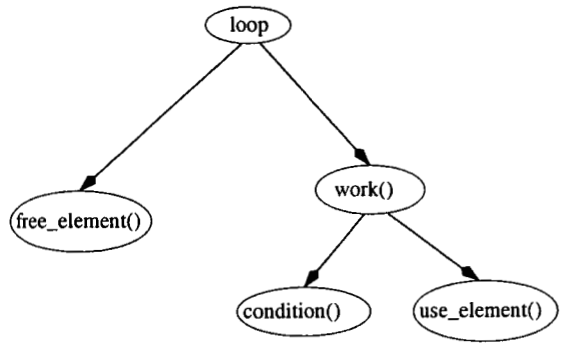
epoch will be read from memory. If *p* happens to be a NULL pointer as well then the program will dereference this NULL pointer just like the original untransformed program did, and cause an exception (depending on the policy of the host operating system).

The size of the *signal_address_buffer* is equal to the number of forwarded values. In practice, the number of values requiring forwarding is small. Our experiments show that we never need a buffer larger than 10-entries.

7.2.3 Compiler Support

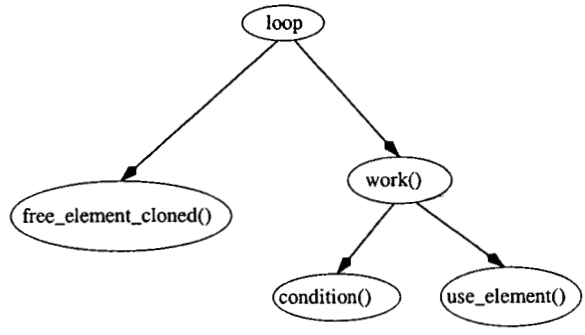
In our approach to TLS support, the compiler is able to both detect and synchronize frequently-occurring data dependences. In this section we demonstrate how the compiler inserts synchronization using the example shown in Figure 7.4. In this example we parallelize a loop that calls the procedures *free_element()* and *use_element()* to add and remove members of a linked list called *free_list*. In every iteration of the loop, the global variable *free_list* is read and modified, potentially causing frequent data dependences and failed speculation unless prevented by proper synchro-

```
void free_element(element) {
   element->next = free_list;          st_1, ld_1
   free_list = element;                st_2, ld_2
}
int use_element() {
   element = free_list;                st_3, ld_3
   free_list = element->next;          st_4, ld_4
   return element;
}
void work() {
   if(condition())                     call_1
      use_element(some_element);       call_2
}
main() {
   do {
      free_element(some_element);      call_3
      work();                          call_4
   } while (test);
}
```



(a) The original program and the corresponding call tree. Function calls, loads and stores are instrumented with labels to identify them.

```
void free_element_cloned(element) {
   f_addr = wait();
   check(f_addr, &free_list);
   f_value = wait();
   m_value = free_list;
   actual_value = select(f_value, m_value);
   resume();
   element->next = actual_value;
   free_list = element;
   signal(&free_list);
   signal(free_list);
}
...free_element(), use_element() and work()
functions omitted for brevity...
main() {
   do_parallel {
      free_element_clone(some_element);
      work();
   } while (test);
}
```



(b) The cloned call tree and the program with synchronization inserted.

Figure 7.4: Compiler-directed procedural cloning and synchronization insertion.

nization. Note that this example is complicated by the fact that the variable *free_list* can be accessed using other names (i.e., *aliases*). Our compiler performs the following steps to synchronize the accesses to this variable:

Profiling dependences: The compiler identifies frequently-occurring, memory-resident, data dependences by profiling all inter-epoch data dependences for each parallelized loop (this profile information is context-sensitive but flow-insensitive). To acquire the profile information, we first associate a unique identifier with each static load and store instruction, and each procedure call point. During execution each load and store instruction can be named by the combination of the instruction identifier and the current *call stack* (the call stack for an instruction, rooted at the parallelized loop, is the list of procedures calls invoked when that instruction is executed). During profiling, each load is matched with any store on which it depends, and the frequency of each dependence is recorded. In Figure 7.4(a), *ld_1*, *ld_3*, *st_2* and *st_4* all access the same memory location denoted by *free_list*, and their dependence relation is illustrated in Figure 7.5. Note that a two memory references with the same identification number but different call stacks are treated separately (i.e., represented by two different vertices in the graph).

Identifying frequently occurring dependences: Unlike scalar values, the same memory-resident value can be accessed with multiple names (through pointer aliasing)—hence we *group* together loads and stores that access the same memory location. It is important to understand that a *group* is different from an alias set. An alias set of pointers is defined conservatively to be a set of pointers that *may* point to the same memory locations. In contrast, (i) pointers in a group will *definitely* access the same memory locations frequently, and (ii) pointers that access the same location might not be grouped if the corresponding data dependences are infrequent.

The compiler chooses groups of pointers by using the dependence profiling information described above to construct a dependence graph, where each load or store instruction with a different call stack is represented by a vertex,

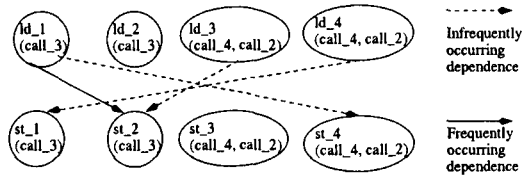


Figure 7.5: An example dependence graph. Each vertex represents a load or store, identified by the combination of a unique number and call stack. Each edge shows a true data dependence between memory references. Ignoring infrequent data dependences, a group is formed with two vertices: *ld_1* and *st_2* (both having call stack *(call_3)*). Accounting for infrequent data dependences would result in an overly-large group.

and each frequently-occurring dependence is represented by an edge. In the resulting graph, each connected component represents a *group*, and all loads and stores belonging to the same group are then synchronized by the compiler as a single entity. Note that we ignore infrequent dependences for performance reasons: if we were to additionally include infrequent data dependences in the graph then our groups would be much larger (as shown in Figure 7.5), leading to over-synchronization and poor performance.

Cloning: For best performance, we want synchronization code to be executed only when necessary to avoid data dependence violations. For example, when a load with a particular call stack is chosen for synchronization, ideally the corresponding synchronization code would only be executed when the load has been reached on a path matching that call stack—the synchronization code should not be executed when the load is reached through some other call path.

Our compiler uses the following steps to implement this code specialization, which basically clones the appropriate procedures on the call stack of a synchronized memory reference. First we build a call tree with the parallelized loop as the root and each call instruction as a decedent of this loop, as shown in Figure 7.4(a). Second, we identify the location in the tree of all frequently-occurring data dependences: for any node containing frequently-occurring dependences, that node and its parents are all cloned, and the original call instructions are modified to refer to these cloned procedures. In our example, the synchronized load and store occurs on the call stack *call_3*, hence the procedure *free_element* is cloned as shown in Figure 7.4(b). Code expansion due to such cloning is negligible (less than 1% on average), since only a small number of procedures are cloned in each application.

Inserting synchronization: *Wait* instructions are inserted before each load instruction to be synchronized, as shown in Figure 7.3(b). However, *Signal* instructions cannot be inserted after every store instruction, since multiple store instructions belonging to the same *group* could occur on a single execution path through an epoch. A signal instruction must occur at least once for each group on every execution path through the epoch, and should occur *after* the last store instruction from that group has been issued. We perform data-flow analyses to find locations that satisfy such constraints to insert the signal operations, similar to the data-flow analyses used to synchronize scalar values [84]. The results of these data-flow analyses are propagated to the cloned procedures to allow signal instructions to be inserted as close as possible to where the value is produced.

7.2.4 Analysis of Data Dependence Patterns

The synchronization mechanism proposed in this section attempts to reduce failed speculation by targeting only frequently-occurring data dependences between consecutive epochs. We now demonstrate that the overall performance penalty due to failed speculation can be mostly attributed to such dependences, and that our decision to ignore infrequent dependences is justified. We performed a limit study using a model with perfect value prediction for loads of interest, which represents an upper bound on the possible performance of synchronizing those loads.

Although it is clear that a frequently-dependent load/store pair should be synchronized, we have yet to experimentally determine a threshold frequency at which synchronization is more beneficial than speculation. To answer this question we conducted the experiment shown in Figure 7.6. First, we identified load instructions that cause inter-epoch data dependences in more than 5%, 15% and 25% of all epochs. Then, we measure the impact of perfect prediction for each set of loads. Although perfect prediction of loads with highly-frequent dependences (eg., 25%) eliminates a significant amount of failed speculation, GZIP_COMP and BZIP2_COMP do not speed up with respect to sequential

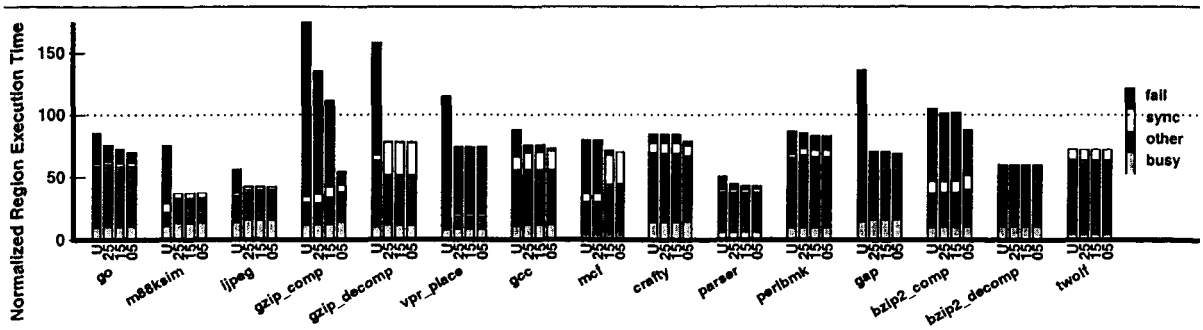


Figure 7.6: Impact of perfectly forwarding value for loads that depend on a store in the previous epoch (dependence distance one), broken down by frequency of the dependence: *U* is the unoptimized case with no forwarding; *25* shows the impact of perfectly forwarding all loads that depend on the previous epoch in more than 25% of all epochs; *15* shows the impact of perfectly forwarding all loads that depend on the previous epoch in more than 15% of all epochs; *05* shows the impact of perfectly forwarding all loads that depend on the previous epoch in more than 5% of all epochs.

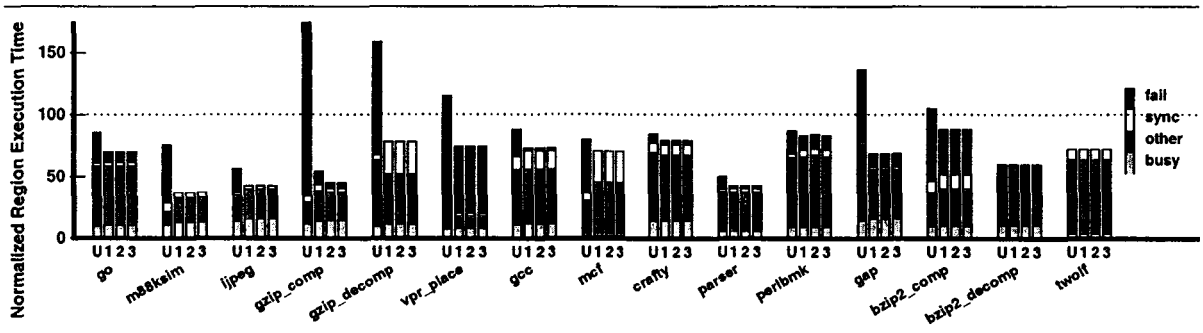


Figure 7.7: Impact of forwarding values for frequently occurring data dependences (5% of the time), broken down by dependence distance: *U* is the unoptimized case with forwarding; *1* shows the impact of perfectly forwarding values for all loads causing dependences of distance one; *2* builds on *1* by forwarding values for loads causing dependences of distance two; *3* builds on *2* by forwarding values for loads causing dependences of distance three.

execution until we additionally predict loads with less-frequently occurring dependences. Only when all loads that cause inter-epoch data dependences in more than 5% of all epochs are perfectly predicted are we able to improve the performance of all benchmarks, suggesting a reasonably low threshold value of 5%.

The distance of a data dependence, in the context of TLS, is the number of epochs between the producer epoch and the consumer epoch. For example, a data dependence between two consecutive epochs has distance of one. To determine the amount of failed speculation caused by dependences of different distances, we carry out an idealized simulation assuming that we can perfectly predict values for loads with dependences of varying distances, as shown in Figure 7.7. We observe that the performance impact for distance-one loads is significant; however, the impact of loads with larger dependence distances is small, and only relevant for one benchmark. Hence dependences of distance one should be the focus of any synchronization effort.

7.3 Infrastructure for TLS

In this section we describe our compiler infrastructure and the underlying hardware support for TLS, as well as our simulation infrastructure and experimental framework.

7.3.1 Compiler Infrastructure

We rely on the compiler to define where and how to parallelize. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [71], and performs the following phases when transforming an application to exploit TLS.

Deciding Where to Parallelize: A *speculative region* is a portion of a program that we speculatively parallelize. In this chapter, we focus solely on loops. With the profile information automatically gathered, the compiler starts with a set of loops chosen to maximize coverage while meeting heuristics for epoch size and loop trip counts: each loop must comprise at least 0.1% of overall execution time and have an average of at least 1.5 epochs per instance, as well as an average of at least 15 instructions per epoch. Loops satisfying these conditions are considered for parallelization. We want to identify the set of loops that are *likely* to minimize total execution time, given that the techniques described in this chapter can improve the performance of value communication through memory. We do so by obtaining an optimistic upper bound on performance by identifying all loads that cause inter-epoch data dependences in more than 5% of all epochs and assume that we can perfectly predict values for these loads during execution. The set of loops that minimize the total execution time of the entire program under this ideal condition are selected for this study. Once loops are selected, the compiler automatically applies loop unrolling to small loops to help amortize the overheads of speculative parallelization. Note that the profiling described above is required only to select which loops are to be parallelized (and not to decide how to forward values). Deciding which regions of code to speculatively parallelize using a minimum amount of profiling information is the subject of ongoing research.

Transforming to Exploit TLS: Once speculative regions are chosen, the compiler inserts new TLS-specific instructions into the code that interact with the TLS hardware to create and manage epochs [66]. For each speculatively parallelized region the compiler inserts explicit synchronization to communicate scalar values between epochs. To avoid parallel epochs being serialized unnecessarily by such synchronization, the compiler schedules instructions within the epoch to reduce the *critical forwarding path* [84].

Inserting Synchronization for Memory-Resident Values: The final optimization step is for the compiler to identify frequently-occurring inter-epoch data dependences through memory resident values and to insert explicit synchronization (the subject of this chapter). In our implementation, we identify these dependences with the help of detailed profile information. The details of this data dependence profiling, as well as the synchronization mechanisms and corresponding compiler support are discussed in section 7.2.3.

Code Generation: Our compiler outputs C source code which encodes our new TLS instructions as in-line MIPS assembly code using `gcc`'s "`asm`" statements. This source code is then compiled with `gcc` 2.95.2 using the "`-O3`" flag to produce optimized, fully-functional MIPS binaries containing these new TLS instructions.

7.3.2 Underlying Hardware Support

The hardware which supports TLS must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation. Our underlying hardware support is based on the scheme proposed in our previous work [66, 67] which extends invalidation-based cache coherence to track data dependences and uses the first-level data caches to buffer speculative state from the rest of the memory system.

7.3.3 Experimental Framework

We evaluate our compilation techniques using a detailed machine model which simulates 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [82], but modernized to have a 128-entry reorder buffer. We simulate a system with four processing cores, where each has its own physically private data and instruction caches, connected to a unified second level cache by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 7.1. We report results for all of the SPECint95 and SPECint2000 benchmarks [23] except for the following: 252.EON, which is written in C++ and therefore not handled by SUIF; 126.GCC, which is similar to 176.GCC; 147.VORTEX, which is identical to 255.VORTEX; 129.COMPRESS, 130.LI, 134.PERL and 255.VORTEX each have low parallel coverage, and hence are not included in the performance graphs.

Table 7.1: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

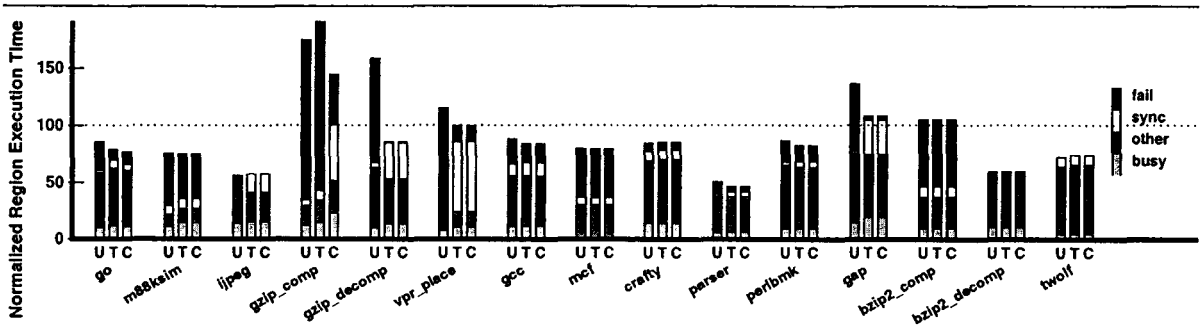


Figure 7.8: Impact of compiler-inserted synchronization using different profiling information. **U** has no synchronization; **T** profiles with the *train* input set and measures with the *ref* input set; **C** profiles with the *ref* input set and measures with the *ref* input set.

For each benchmark, after skipping over the initialization phases, we simulate approximately a billion instructions using the first input in the `ref` input set.

7.4 Performance Evaluation

We now present the results of our experiments to quantify the performance impact of our compiler-based technique, and we also compare it with related hardware-based techniques [21, 68].

7.4.1 Impact of Compiler-Inserted Synchronization

Figure 7.8 shows the performance impact of our compiler algorithm (described earlier in Section 7.2) for synchronizing frequently-occurring memory-resident data dependences. This figure shows the time spent in parallelized regions of the code (we will focus on overall program speedups later in Section 7.4.3), normalized to the time spent in those regions in the original sequential program. (Hence if a bar is less than 100, it means that the parallelized regions would be speeding up under TLS.) The **U** bars are the baseline ("unsynchronized") case that we are attempting to improve upon: they contain no synchronization for memory-resident values (but may contain synchronization for scalar register values [84]). The **T** and **C** bars show the impact of compiler-inserted synchronization for memory-resident values on region performance with the `ref` input sets, where profiling was done with the `train` (**T**) and `ref` (**C**) input sets, respectively.

Comparing **C** with **U** that compiler-inserted synchronization improves performance in half of the benchmarks (GO,GZIP_COMP, GZIP_DECOMP, VPR_PLACE, GCC, PARSER, PERLBMK, and GAP), and has no significant impact in the other seven cases. (Note that in two of the seven cases where our technique did not improve performance—BZIP2_DECOMP and TWOLF—failed speculation was not a problem to begin with.) Among the seven cases that do improve, the amount of execution time wasted on failed speculation (*"fail"*) is reduced by an average of 68%. Although some of this gain is offset by an increase in time stalled waiting for synchronization (*"sync"*), these seven

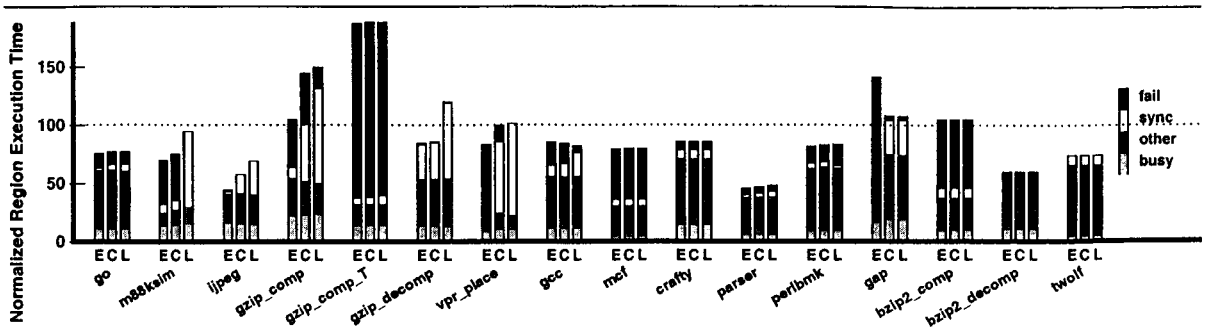


Figure 7.9: Potential impact of changing to a more aggressive or less aggressive synchronization scheme. **E** is the idealized case where the consumer can perfectly predict synchronized memory values; in **C** the consumer stalls any load of a synchronized memory value until the producer forwards the value; in **L** the consumer stalls any load of a synchronized memory value until the producer completes.

applications still enjoy an average region speedup of 17%.

By comparing the **T** and **C** bars in Figure 7.8, we can get a sense of how the accuracy of the profiling information used by our compiler can affect the quality of its results. The **T** bars represent the more realistic scenario where profiling is done with a different input set (`train`) than the one used in the actual run (`ref`), and the more optimistic scenario for the **C** bars (where the profiling and actual input sets are the same) is included for the sake of comparison. Note that in all but one case (GZIP_COMP), the results are fairly insensitive to the choice of profiling input set. In GZIP_COMP, however, the flow of control is complex and sensitive to the input set, and this in turn determines which loads and stores are dependent; hence different profiling input sets can lead the compiler to synchronizing different pairs of loads and stores. Because the **T** and **C** cases behave differently for GZIP_COMP, we will present both cases throughout the remainder of this chapter as "GZIP_COMP_T" (**T**) and "GZIP_COMP" (**C**).

By synchronizing data dependences, we trade time spent on failed speculation with that on synchronization. When synchronization is not properly placed, it could create a critical forwarding path which dominates execution time. Although reducing the cost of synchronization is not the goal of this work, we attempt to evaluate the significance of synchronization with two experiments: in Figure 7.9, the **E** bars correspond to an idealized experiment where the consumer is always able to perfectly predict any synchronized memory value. This eliminates all time spent on synchronization of memory values, but may increase violations since it increases parallel overlap. The **L** bars in Figure 7.9 correspond to a more conservative forwarding scheme where synchronized loads issued by the consumer are stalled until the previous epoch completes.

For M88KSIM, IJPEG, GZIP_COMP, GZIP_DECOMP and VPR_PLACE, execution time is positively correlated with the cost of synchronization. This indicates that stalling frequently violated loads until previous thread completes could serialize the execution unnecessarily and degrades performance. On the other hand, being able to forward the value early can reduce synchronization and improve performance.

7.4.2 Comparison with Hardware-Inserted Synchronization

Previous research [21, 68] proposed two *hardware* techniques to reduce the cost of failed speculation due to memory-resident values: *prediction* and *synchronization*. Neither of the proposed techniques require centralized structures to match dependence pairs; however, the they differ in complexity, from a 2KB violation prediction table [21] to two 32-entry tables that track loads which are exposed and loads which have caused speculation to fail [68]. We have implemented hardware-based *prediction* and *synchronization* as described *et. al* [68]. In Figure 7.10, we compare our compiler-inserted synchronization techniques with these two hardware mechanisms. The **P** bar shows that the *value prediction* technique that we have evaluated has insignificant effect on performance, indicating that forwarded memory-resident values are unpredictable. In the rest of this section, we focus on comparing hardware-inserted synchronization (**H**) with compiler-inserted synchronization (**C**). For the hardware inserted synchronization, the hardware identifies loads that frequently cause violations and stalls these loads until the previous epoch completes. To avoid over-synchronization of infrequently-dependent loads, we periodically reset the table that tracks the loads that have caused speculation to fail.

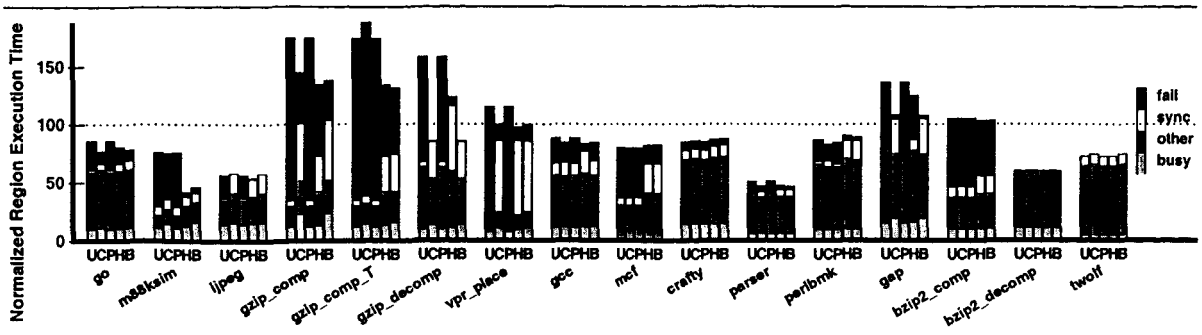


Figure 7.10: Comparison of compiler-inserted synchronization, hardware-inserted synchronization, and a hybrid scheme. **U** has no synchronization; **C** uses compiler-inserted synchronization; **P** uses hardware value prediction; **H** uses hardware-inserted synchronization with the violating load table periodically reset; **B** uses a hybrid of compiler-inserted and hardware-inserted synchronization with the violating load table periodically reset.

A comparison between compiler-inserted and hardware-inserted synchronization reveals that *each of the techniques wins in some cases but none of them wins for them all.* In eleven out of the fifteen benchmarks, at least one synchronization technique is able to improve the performance over the unoptimized case. Four benchmarks, GO, GZIP_DECOMP, PERLBMK and GAP, achieve the best performance with compiler-inserted synchronization; three benchmarks, M88KSIM, GZIP_COMP, VPR_PLACE, achieve the best performance with hardware-inserted synchronization. For the rest of the benchmarks, the two techniques are comparable. Here we attempt to explain why each benchmark responds differently to the two optimization techniques:

- In M88KSIM, violations are not caused by true data dependences, rather they are caused by false sharing. The compiler is attempting to synchronize true dependences, while the hardware is tracking dependences at a cache line granularity. Since violations are tracked at a cache line granularity, the hardware inserted synchronization yields the best results—we could track dependences at a cache line granularity in the compiler as well, but we believe that other techniques (such as memory layout optimizations or loop unrolling) are better for addressing false sharing in the compiler.
- In GZIP_DECOMPRESS, the compiler and the hardware both insert synchronization, however, the compiler is able to speculatively forward the desired value much earlier than our hardware can. This avoids over-synchronization, resulting in better performance.
- Software-inserted synchronization can be conservative—it synchronizes dependences which *may* or *may not* actually happen at runtime, depending on the timing of the epochs. If a load tends to be executed only when all prior epochs have completed, then it will rarely cause a violation. In such a case, the synchronization code just adds extra overhead— this is the cause of the small performance degradation in TWOLF.

Since the hardware and the compiler based synchronization can each benefit a different set of benchmarks, we conduct the following experiment to determine whether the two techniques are synchronizing the same set of memory-resident values: we invoke our synchronization scheme to mark each load instruction as *would be* synchronized by the compiler and/or by the hardware (depending on the execution mode, we may or may not stall for marked synchronization). When a violation does occur, we record whether the load that caused this violation *would* have been synchronized, and by which scheme. We execute the program under four different modes, and show the results in Figure 7.11: (i) do not stall for any synchronization, denoted by the **U** bars; (ii) only stall for compile-inserted synchronization, denoted by the **C** bars; (iii) only stall for hardware-inserted synchronization, denoted by the **H** bars; (iv) stall for both hardware-inserted and compiler-inserted synchronization, denoted by the **B** bars. We observe that a significant number of violating loads would only be synchronized by either the hardware or the compiler, but not both. Our existing compiler and hardware support can be complementary as follows: (i) the hardware synchronizes violating loads that are not identified by profiling information; (ii) the compiler reduces the cost of synchronization by providing the forwarded value early. Two possible ways to further enhance complementary behavior are (iii) for the hardware to filter out compiler-inserted synchronization that rarely forward the correct values; and (iv) for the hardware to reset a violating load less frequently if the compiler hints that it will occur frequently.

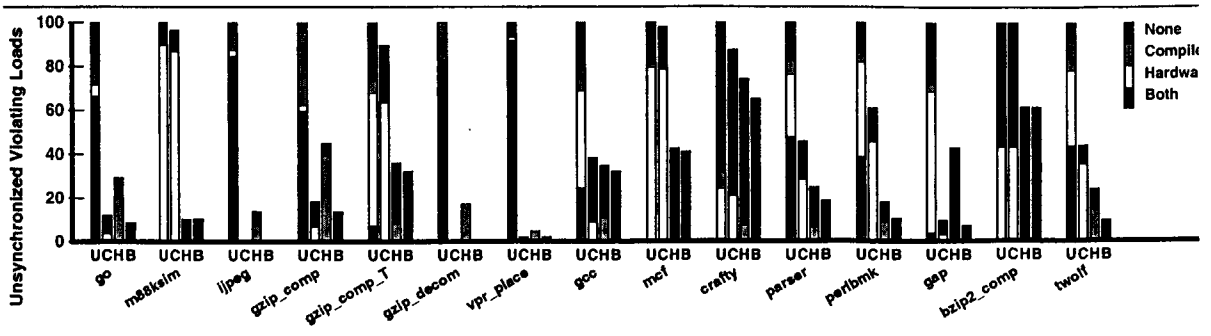


Figure 7.11: Breakdown of all loads that cause violations by whether they would be synchronized by hardware or compiler-inserted synchronization. **U** stalls on no synchronization; **C** stalls only on compiler-inserted synchronization; **H** stalls only on hardware-inserted synchronization; **B** stalls on both compiler-inserted and hardware-inserted synchronization. BZIP2_DECOMP is omitted because it has no violations.

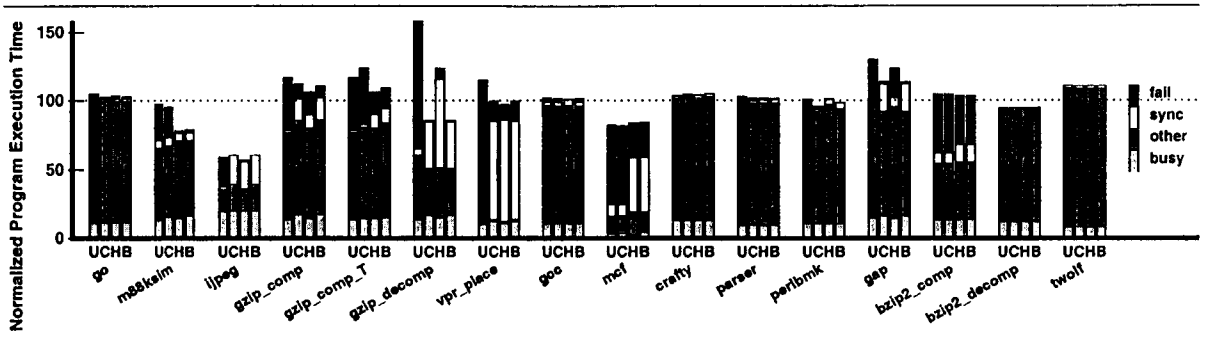


Figure 7.12: Program speedup. **U** is the case with no synchronization; **C** is compiler-inserted synchronization; **H** is hardware-inserted synchronization with the violating load table periodically reset; **B** is a hybrid of compiler-inserted and hardware-inserted synchronization.

To illustrate the feasibility of such a compiler-hardware hybrid, we enable both hardware synchronization with periodic reset and compiler-inserted synchronization. The results are shown in Figure 7.10 as bar **B**. In several benchmarks, the hybrid approach nearly captures the performance of the best of the two techniques: M88KSIM benefits from hardware-inserted synchronization and avoids the cost of false sharing, while GZIP_DECOMP benefits from having values be forwarded early by compiler-inserted synchronization. Therefore, it is possible for us to implement a hybrid that can improve the performance of a larger set of programs by taking advantage of both compiler and hardware inserted synchronization.

7.4.3 Program Performance

Since we are interested in studying the impact of synchronization on parallelized code, we so far have focused on region speedups. In Figure 7.12 we instead take the coverage of these loops into account and look at the performance impact on the whole program. We see that inserting synchronization of memory values has a significant positive impact for six of these benchmarks, and that the best results overall can be achieved with a hybrid of both software and hardware synchronization. Table 7.2 presents the speedups in detail, and we see that relatively large speedups in our parallel regions are sometimes offset by slowdowns in our sequential code. (Ideally, we should see a speedup of 1.0 in the sequential regions.) This is a side effect of our compiler infrastructure—the inline assembly we use to instrument parallelized loops can inhibit the optimization and register allocation of our `gcc` back-end, causing this measurement artifact. This overhead remains constant regardless of the hardware and/or compiler optimizations applied. We anticipate that with a proper compiler back-end (instead of using a source-to-source compiler followed by `gcc`) even better program performance would be observed.

Table 7.2: Region coverage and program speedup (relative to sequential execution).

Benchmark	Coverage	Parallel Region Speedup		Sequential Region Speedup		Program Speedup	
		Compiler-only	Both	Compiler-only	Both	Compiler-only	Both
099.go	22%	1.29	1.27	0.90	0.90	0.97	0.97
124.m88ksim	56%	1.33	2.15	0.82	0.82	1.04	1.25
132.ijpeg	97%	1.73	1.73	0.52	0.52	1.64	1.64
164.gzip_comp	25%	0.69	0.72	0.98	0.98	0.88	0.90
164.gzip_comp_T	25%	0.52	0.75	0.98	0.98	0.80	0.91
164.gzip_decomp	99%	1.16	1.16	0.93	0.93	1.16	1.16
175.vpr_place	99%	1.00	1.00	0.97	0.97	1.00	1.00
176.gcc	18%	1.18	1.18	0.94	0.94	0.98	0.98
181.mcf	89%	1.25	1.21	0.99	0.99	1.21	1.18
186.crafty	14%	1.16	1.13	0.92	0.92	0.95	0.95
197.parser	37%	2.13	2.14	0.74	0.74	0.98	0.98
253.perlbmk	20%	1.20	1.12	1.00	0.98	1.04	1.01
254.gap	57%	0.92	0.92	0.82	0.82	0.87	0.88
256.bzip2_comp	63%	0.94	0.96	0.96	0.96	0.95	0.96
256.bzip2_decomp	13%	1.66	1.66	0.99	0.99	1.05	1.05
300.twolf	19%	1.34	1.34	0.84	0.83	0.90	0.90

7.5 Summary

TLS provides a mechanism for speculating that data dependences across optimistically-parallelized threads do not exist. Like most forms of speculation, however, when you speculate correctly you win, but when you speculate incorrectly, you can actually hurt performance. Hence an important question is *how frequently* do inter-thread data dependences occur? If they occur frequently enough for a given load-store pair, we may be better off explicitly synchronizing the threads so that the consumer waits for the value (or at least a good guess of what the value might be) from the producer. In previous work, we demonstrated that compiler-inserted synchronization for *scalar register* values was an important technique for improving TLS performance [84], and in this chapter we tackled the question of whether the same is also true for *memory-resident* values.

We observe that for most benchmarks failed speculation is usually caused by load instructions that suffer dependence violations relatively *frequently* (e.g., at least 25% of the time), which makes them easy to spot given a mechanism for profiling data dependences. However, for some benchmarks we must be able to synchronize data dependences that only occur in 5% of the epochs to achieve a reasonable speedup. In addition, we also observe that the producer and consumer of inter-thread dependences are usually *consecutive* epochs, which simplifies the process of explicitly forwarding the data.

Our performance results demonstrate that applying compiler-inserted synchronization to *memory-resident* values that would otherwise cause frequent dependence violations does improve TLS performance in many cases: half of the applications enjoyed significant region speedups, while the other half were unaffected. The most dramatic case was GZIP_DECOMP, which went from a significant region slowdown to a significant region speedup using our technique. For GZIP_DECOMP and several other cases, we observe a significant benefit from the compiler's ability to forward data when it is produced (rather than waiting until the producer thread completes its execution).

Comparing our compiler-based approach with a hardware-based approach to synchronizing memory-resident values, we observe that both approaches are useful, and that neither approach consistently dominates the other: sometimes the compiler-based approach is much better than the hardware-based approach, and vice-versa. We observe that the two different approaches seem to behave differently because they often choose different sets of load instructions to synchronize. This suggests that a hybrid approach that combines the advantages of both approaches might be best. While the simple hybrid approach that we explored did not outperform the best of the two approaches for a given benchmark, it did a better job of tracking the best performance overall than either approach individually. In future work, it may be possible to design an even better hybrid approach.

Chapter 8

The Impact of Thread Size and Selection

8.1 Introduction

The choice of speculative threads for TLS can have a significant impact on performance. In this study we limit our focus to loops, for two reasons: (i) loops comprise a significant portion of execution time (the loops studied here have an average coverage of 87.9% across 25 Spec [24, 25] benchmarks); (ii) loops are regular and predictable, easing compiler analyses. In Figure 8.1, we contrast a naïve thread selection scheme that chooses loops to parallelize by maximizing coverage with a more intelligent scheme that parallelizes selectively and applies loop unrolling. Evidently, wise thread selection can be the difference between a speculatively parallelized application slowing down and speeding up.

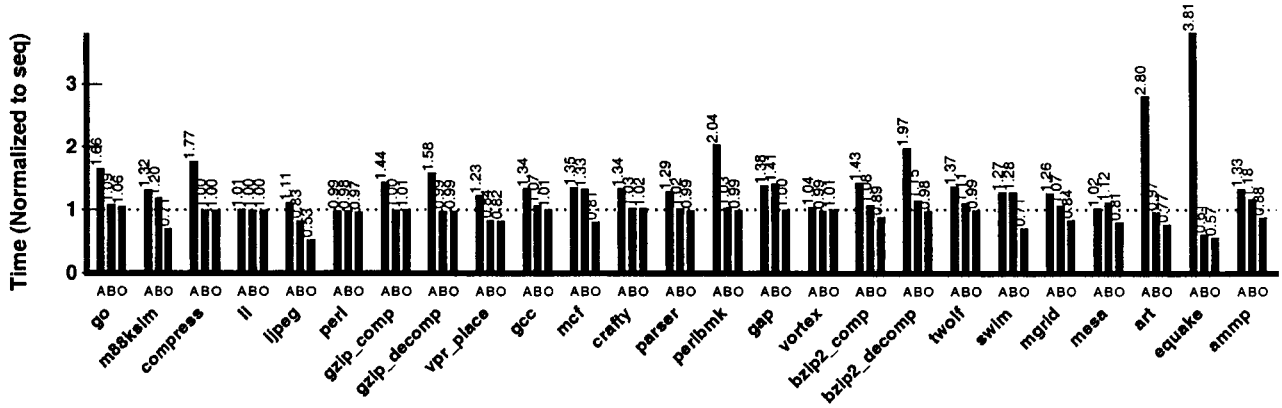


Figure 8.1: Performance impact of thread size and selection. The *A* bars show performance where coverage is maximized and no unrolling is applied; the *B* bars apply static global loop selection without unrolling; the *O* bars show static global loop selection with unrolling applied.

For each loop in a program, there are multiple ways we can partition the iterations into speculative threads through loop unrolling. While we do not find clear rules for this partitioning, we find that there is a tension between choosing larger and smaller threads. Increasing thread size can amortize the overheads of thread creation, can eliminate dependences caused by false sharing, and can reduce the cost of synchronization. However, larger threads can have more frequent data dependence violations, can reduce parallelism by decreasing the total number of threads (i.e., to less than the number of processors), can amplify load imbalance, and can increase the amount of speculative buffering required as well as the cost of recovering from failed speculation. In Section 8.2 we see that this tension produces surprising results.

Once we understand how to partition each loop into speculative threads, we then have to decide *which* speculative threads should be selected for parallel execution: e.g., are the larger threads in an outer loop better than the smaller threads in an inner loop? We examine this question in Section 8.3, and find that the loop selection decision is crucial for achieving good performance. We find that significant profile information is needed to perform loop selection statically at compile time, but that local decisions can match the performance of whole program analysis. In Section 8.4 we develop a mechanism for delaying the loop selection decision until run time, using performance counters to make this decision dynamically.

8.1.1 Related Work

This work builds upon several years of research on TLS hardware support [3, 20, 37, 39, 40, 44, 48, 59, 67, 73] and compiler support [77, 79, 84].

The most extensive prior study of speculative thread selection was for the Multiscalar compiler [76], where heuristics and execution-frequency profiles are used to merge basic blocks into *tasks*. The authors find that small tasks perform well on the aggressive and tightly-coupled Multiscalar architecture: their heuristics do not select tasks containing inner loops that are too large to be fully unrolled, nor function calls that are too lengthy to be inlined.

Through evaluation on a simple architectural model, Oplinger *et al.* [59] conclude that loops contain significant speculative parallelism, but that there is additional benefit from procedure continuations and by simultaneously executing multiple loops in parallel.

Marcuello *et al.*'s speculative thread spawning scheme [50] (and corresponding hardware support) emphasizes flexibility: a program may be partitioned into threads arbitrarily, and a thread may be spawned from any point. A run-time filter eliminates poorly-performing threads, and an aggressive value predictor minimizes the impact of inter-thread register dependences—leading to a thread selection heuristic that minimizes control mis-speculation but considers neither data dependences nor load balance.

8.1.2 Contributions of this Chapter

This chapter makes the following contributions. First, we consider 1462 loops with a wide range of speculative thread sizes—including threads containing lengthy inner loops and procedure calls. We focus solely on loop-level parallelism, and our in-depth evaluation of compiler techniques is based on a detailed architectural simulation. Second, since our compiler minimizes critical forwarding paths and applies loop unrolling, we find—in contrast with Oplinger *et. al*'s work [59]—that parallelizing *doacross loops* [26] is profitable. Third, we explore run-time techniques for dynamically selecting speculative threads (similar to Marcuello *et al.* [50]), but by limiting ourselves to loops we simplify the compiler analyses for inserting synchronization and scheduling instructions—which in turn eliminate the need for hardware value predictors [84].

8.2 Loop Unrolling

An important factor in the effectiveness of TLS is the size of the speculative threads.[1] When a loop is divided into threads at loop boundaries, both loop unrolling and strip mining can be used to combine multiple loop iterations, producing larger threads. In this chapter we apply loop unrolling to increase thread size; strip mining would likely yield similar results. We find that unrolling a loop not only increases the size of threads, but may also impact the frequency and cost of failed speculation, the performance of synchronization, and the amount of available parallelism.

8.2.1 Unrolling Study Methodology

In this chapter, we evaluate the impact of unrolling on a large set of loops by applying unrolling factors of 1 (no unrolling), 2, 4 and 8 to the loops and then simulating their sequential and parallel execution. We will evaluate the results of this study in Section 8.2.2, but first we describe our compilation and simulation infrastructures.

[1] Note that the speculative threads discussed in this chapter are not necessarily bound to architecturally-visible threads: a single architectural thread may execute multiple speculative threads.

```
do{
   work();
} while(test);
```
(a) Original loop.

```
if(!is_parallel) {
   is_parallel = true;
   do_parallel{
      work();
   } while(test);
   is_parallel = false;
} else{
   do{
      work();
   } while(test);
}
```
(b) Transformed for parallel execution.

Figure 8.2: Loop transformation for TLS execution. The `is_parallel` flag prevents recursive invocations of the parallel loop.

Compiler

Our compiler is based on the SUIF 1.3 compiler system. For `Fortran` benchmarks, source files are first converted to `C` using `sf2c`, and then all `C` files are then converted to SUIF format. Our compiler selects loops and parallelizes them, inserting new instructions to manage speculative threads. The compiler outputs `C` code—with the TLS instructions encoded as MIPS assembly code using `gcc`'s `asm` primitive—which is then compiled with `gcc` 2.95.2 using the `-O3` flag to produce optimized, fully-functional MIPS binaries containing new TLS instructions.

In this study we only parallelize the outer-most dynamic instance of a selected loop. This is achieved by using a global flag to guard each parallelized loop, transforming the loop as shown in Figure 8.2. The parallel loop is then unrolled. Unrolling of `for` loops results in an unrolled for loop followed by a wind-down loop; `do-while` loops are unrolled with conditional `break` statements after each iteration.

Primitives are inserted into the code to fork threads, and to mark the start and end of the speculative code in each thread. The `setjmp` and `longjmp` functions are instrumented to check whether a parallel execution is being exited. All local scalars that are communicated between threads are synchronized using primitives for value forwarding. Since this synchronization introduces a critical forwarding path which can serialize execution, we apply a dataflow based scheduling algorithm [84] to minimize the impact of the critical forwarding path.

Benchmarks

We study the SpecInt95 and SpecInt2000 benchmarks which compile with SUIF 1.3. We also study the six SpecFP2000 benchmarks written in `Fortran77` (since conversion to SUIF format of `Fortran90` codes fails).

For all appropriate benchmarks we also skip the initialization portion of execution and begin simulation with a "warmed-up" memory system, loaded from a pre-saved snapshot of the cache contents. We use the `ref` input sets of the benchmarks, as detailed in Table 8.1. We simulate up to the first billion user-mode instructions. Since the sequential and TLS versions of each benchmark are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code so that the executions are comparable.

The benchmarks provide a large set of loops to evaluate. We study all of the loops that meet the following criteria:

- Coverage is larger than 0.1% of execution time. (This eliminates insignificant loops.)
- Loop iterates more than once. (Ignore loops with no potential parallelism.)
- Average iteration size is less than 16k instructions. (Overly large loops are not likely to work with TLS.)
- Total instructions contained in each invocation of the loop is greater than 30. (We need enough instructions to make multiple threads from.)
- Loop did not contain a call to `alloca()` (which would interfere with our stack management code.)

1462 loops meet the above criteria; the coverage (fraction of instructions executed) of the outermost loops is quite high for most benchmarks (see Table 8.1). A few benchmarks suffer from low coverage: this is due to either large recursive sections of execution (e.g., the game tree search in CRAFTY) or due to loops formed from `goto` statements which are not recognized as being structured loops by SUIF (the Yacc parsers in PERLBMK and PERL), or large non-looping sections of code (VORTEX).

Benchmark		Input	Loop Coverage	Pruned Coverage
SpecInt95	go	9stone21.in from ref	100%	99.9%
	m88ksim	ref	100%	99.54%
	compress	reduced ref input (5.6MB)	100%	100%
	li	ref	100%	99.65%
	ijpeg	vigo.ppm from ref	100%	99.52%
	perl	primes.pl from ref	5.61%	1.58%
SpecInt2000	gzip_comp	input.source, comp phase	100%	100%
	gzip_decomp	input.source, decomp phase	100%	99.97%
	vpr_place	place portion of ref input	100%	99.97%
	gcc	expr.i from ref	97.6%	94.66%
	mcf	ref	100%	100%
	crafty	ref	100%	40.77%
	parser	ref	100%	99.41%
	perlbmk	diffmail.pl from ref	62.99%	59.74%
	gap	ref	100%	97.89%
	vortex	bendian1.raw from ref	99.18%	15.86%
	bzip2_comp	input.source, comp phase	100%	99.97%
	bzip2_decomp	input.source, decomp phase	100%	100%
	twolf	ref	100%	99.96%
SpecFp2000	swim	ref	100%	99.96%
	mgrid	ref	100%	99.93%
	mesa	ref	100%	99.81%
	art	first ref input	100%	99.99%
	equake	ref	100%	100%
	ammp	ref	100%	89.55%

Table 8.1: Inputs used for each benchmark.

Table 8.2: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16kB, 8 history bits)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32kB, 4-way set-assoc
Data Cache	32kB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

We measure the performance of each loop in isolation when executed both sequentially and speculatively-in-parallel for unrolling factors of 1 (no unrolling), 2, 4 and 8.

Simulator

TLS hardware support must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation. The scheme we implement *et. al.* [67] uses the first-level data caches as speculative buffers and an extended version of standard invalidation-based cache coherence to track data dependences. While we evaluate our compiler support on this specific implementation of TLS, we expect that our conclusions would be similar for other TLS hardware implementations [3, 20, 37, 39, 40, 44, 48, 59, 67, 73].

We evaluate our compilation techniques using a detailed machine model which simulates four 4-way issue, out-of-order, superscalar processors similar to the MIPS R14000 [82], but modernized to have a 128-entry reorder buffer. Each processor has its own physically-private data and instruction caches, connected to a unified second level cache by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 8.2. Threads up to 200k dynamic instructions in length are executed in parallel, and longer threads are executed sequentially—filtering of these threads from the results caused 54 of our 5872 loop measurements to be omitted.

Measurements

Starting with performance data for each loop when run in parallel and sequentially with unrollings of 1, 2, 4, and 8 times we consider two metrics: *speedup* and *gain*. In this chapter we define the *speedup* of a loop, S_l to be the ratio of the *best* sequential execution time (of all the unrollings) and the parallel execution time of the loop. The *gain* of a loop, G_l, is the fraction of the total program execution time saved by parallelizing loop l, and hence takes the *coverage* C_l of a loop into account: $G_l = C_l(1 - \frac{1}{S_l})$. A loop gain of 0.05 would mean that parallelizing that loop would result

in a program that eliminated 5% of execution time.

8.2.2 Unrolling Study Results

We start with measurements of 1462 loops, and try to categorize these loops according to how unrolling affects their performance when run using TLS. As examples we use the loops which benefit the most from unrolling: those with a gain (reduction in program execution time) of more than 1% when the best unrolling is applied, and where the change in gain due to unrolling is larger than 1%: these loops are shown in Figure 8.3.

In Figure 8.3 we show the time spent executing each loop for each unrolling. In the following sections we will see that the change in performance due to unrolling can be broken down into four independent segments, which we measure by categorizing graduation slots: *sequential* represents time spent either running code outside of epochs (such as the wind-down loop) or time spent idling due to insufficient parallelism; *sync* time is time spent stalling awaiting the arrival of a forwarded value from a previous thread; *violation* represents time spent executing code which is then undone by a violation; *busy* represents time spent executing code in the parallel region which successfully commits. These four segments can be considered separately, and will guide the design of a compiler pass to find the best unrolling.

Available Parallelism

Unrolling increases the size of the threads formed from a loop, while reducing the total number of threads. If a loop does not have many iterations to start with this can be harmful. In Figure 8.4 we see that a large number of the loops in our benchmarks have a very small trip count. Unsurprisingly, the loops with average trip counts of less than two never speed up when TLS is applied since they lack parallelism. In loops with low trip counts, we have found that unrolling has no practical benefit since the reduction in parallelism is always harmful to performance. In Graph 17 in Figure 8.3 we see that with higher unrolling factors the time spent executing sequentially starts to dominate—this is caused by a lack of parallelism.

Overhead

If the body of a loop is small, then the overhead associated with spawning threads and communicating initial values between threads may dominate execution, making TLS execution of the loop not worthwhile. Unrolling amortizes that overhead.

In Figure 8.3 we see this as a reduction in busy time. About a third of the graphs show this quite prominently — busy time is reduced dramatically as unrolling is applied. These cases all have small loop bodies, suffer from few violations and are not constrained by synchronization. In a few cases the reduced overhead from unrolling is offset by other effects, such as in Graph 17 where a lack of available parallelism limits the gain of reduced overhead.

Synchronization

Our TLS scheme uses producer-consumer synchronization to communicate frequently-used values between threads. When unrolling is applied to a loop containing synchronization, performance can be severely degraded since the length of the critical forwarding path is increased, as illustrated in Figures 8.5(a) and 8.5(b). In such cases, we depend on our compiler's instruction scheduling pass [84] to reduce the size of the critical forwarding path by moving non-critical code outside of the wait/signal pair, resulting in the improved execution illustrated in Figure 8.5(c). However, unrolling can actually increase the length of the critical forwarding path whenever scheduling is blocked by ambiguous data dependences or control flow. In contrast, unrolling can also have a positive effect on synchronization: for example, unrolling by a factor of two halves the number of synchronization operations while doubling thread size. Halving the number of synchronization operations reduces time spent waiting for forwarded values, while increasing thread size gives our out-of-order processors a larger instruction window from which to extract ILP. In addition, synchronization is only a bottleneck if there is insufficient unsynchronized computation in the loop to overlap with the synchronized computation and communication: reducing synchronization can remove this bottleneck.

In Figure 8.3 we see that most loops which improve as the unrolling factor increases do not spend significant time stalled on synchronization. In most cases where time *is* spent stalled on synchronization, that time does not increase much with unrolling—indicating that instruction scheduling is effective. One exception is Graph 22 where synchronization time increases dramatically when unrolling is applied, overwhelming any potential gains from the reduction of overhead.

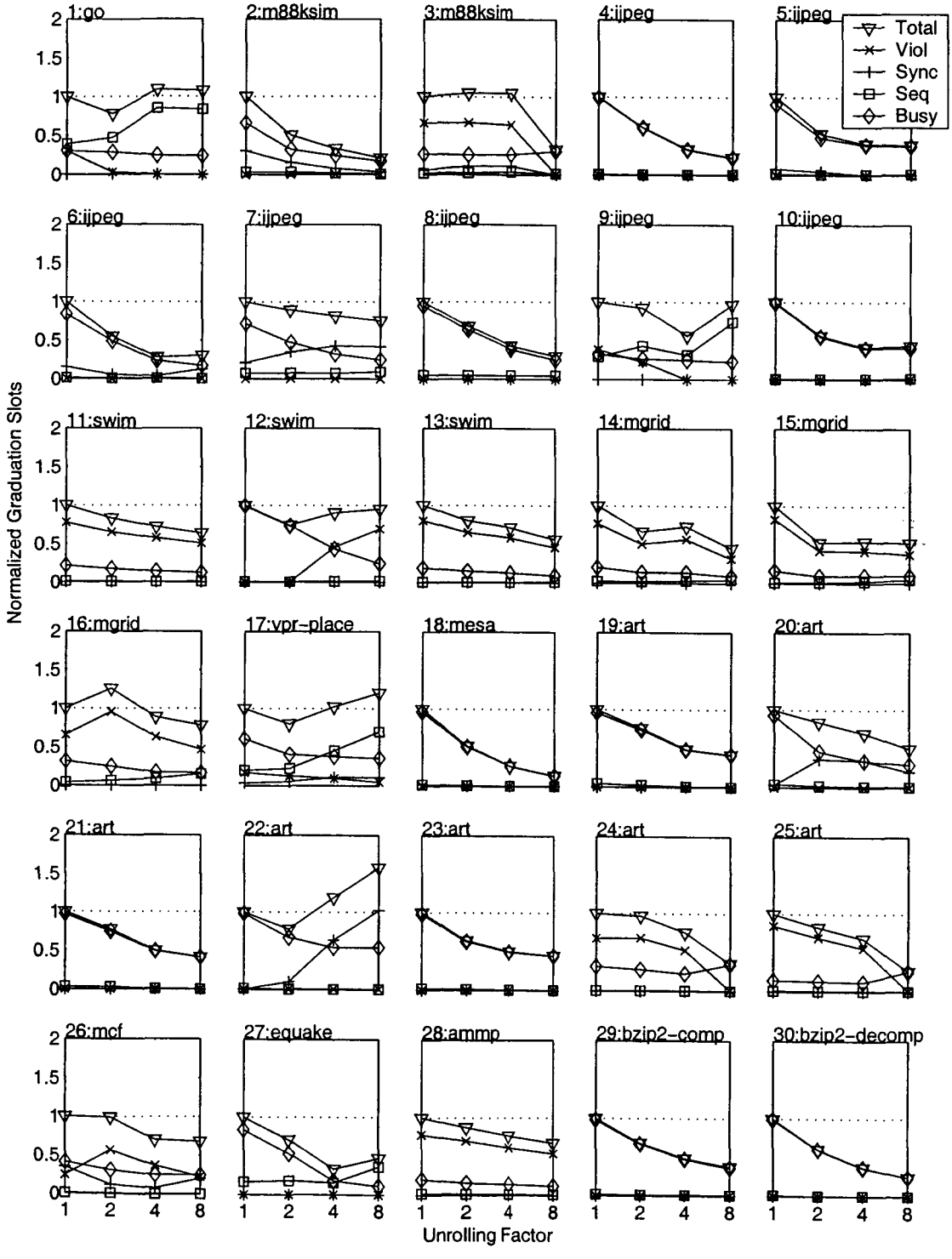


Figure 8.3: The loops where unrolling has the largest impact: the gain is greater than 1% and the change in gain due to unrolling is more than 1%. The upper line shows total performance impact of unrolling; the lower lines show a breakdown into components. [A color version of this graph can be found at: `http://www.cs.cmu.edu/~colohan/papers/hpca04/top30.eps`]

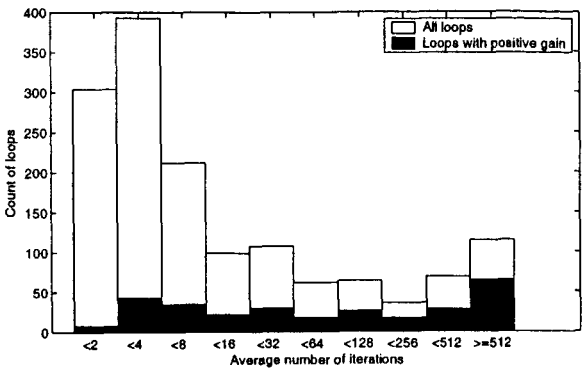


Figure 8.4: There are many more loops with small trip counts than with large trip counts in the benchmarks, but large trip count loops are more likely to benefit with TLS.

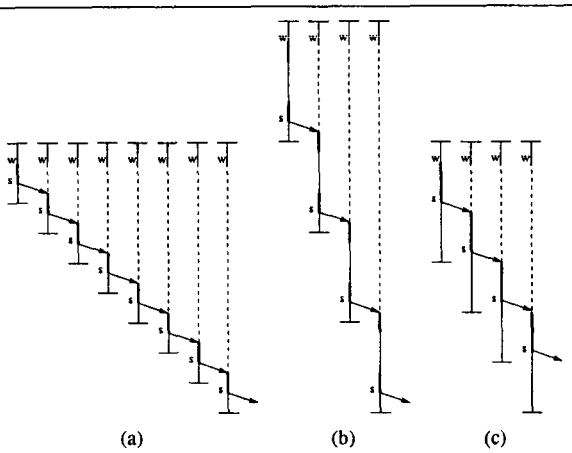


Figure 8.5: The impact on the performance of synchronization of unrolling by a factor of two: (a) before unrolling; (b) after unrolling; (c) after scheduling; A "w" represents a *wait* primitive and a "s" represents a *signal* primitive. Note the reduction in the length of the critical forwarding path from the first wait to the last signal between (a) and (c).

Failed Speculation

Loop unrolling can also affect the amount of failed speculation by altering the patterns of memory references and inter-thread data dependences. A loop with iterations that access consecutive memory locations (e.g., stepping through an array) may suffer from failed speculation due to false sharing, since our implementation of TLS hardware support tracks data dependences at a cache line granularity. When such a loop is unrolled the proper amount, the memory accesses can be grouped so that thread boundaries match cache line boundaries, eliminating false sharing and reducing the frequency of failed speculation.[2] When loop iterations are periodically dependent (e.g., every other pair of iterations is dependent) then unrolling can reduce failed speculation by making these dependences internal to a single iteration/thread. When a loop accesses memory randomly (e.g., updating a hash table) then unrolling increases the probability of inter-thread dependences and hence failed speculation.

We see a few instances of unrolling eliminating false sharing in Figure 8.3, in Graphs 3, 24 and 25. Graph 12 has a good example of the trade off associated with unrolling: unrolling reduces the overhead of TLS while increasing the fraction of threads that suffer from violations.

[2] An alternative that we have not yet evaluated is to pad data structures to align accesses to cache line boundaries.

8.2.3 Choosing the Right Unrolling

Given the potential downsides of unrolling—limited parallelism, overhead, failed speculation and synchronization—choosing the proper unrolling for a loop can be a challenge. We offer the following advice based on our observations.

First, we assume that we only need to find the best unrolling for those loops for which the parallel version speeds up with respect to the sequential version (i.e., the loops with positive *gain*); when filtering the measured set of 1462 loops to include only these loops with positive gain, 292 loops remain. Next, unrolling a loop with a low trip count (eg. less than 4) will result in idle processors; when we filter such loops, 241 loops remain. By performing this aggressive filtering we can evaluate trends in the loops that really matter for TLS, and avoid noise from the other data.

Thread overhead dominates loops with small bodies. We found that in general loops with 30 instructions or fewer per iteration did not benefit from speculative parallelism without unrolling: of our 241 loops, 46 loops fall into this category. In addition, when loop bodies are small we found it best to unroll as much as possible: 44 of these 46 loops perform best when unrolled by a factor of 8 (the other two perform slightly better with lower unrolling factors). Since we have decided to unroll these small loops, we filter out these 46 loops from further consideration and continue with 195 loops.

The compiler is able to estimate the length of the critical forwarding path, as well as the corresponding impact of unrolling and scheduling—hence the compiler can curtail unrolling when the estimated impact is negative. 65 of our remaining 195 loops spend more than 10% of their time waiting on synchronization when unrolled by a factor of 8.

If the presence of a sequential access pattern in a loop is detected by the compiler then it can apply unrolling to eliminate false sharing effects. We found that 26 of our remaining 195 loops reduce their time spent on violations significantly when unrolled (change from losing more than 10% to less than 1% of their time to failed speculation). Non-sequential accesses are unpredictable, and when they are present the compiler can often not decide what the best unrolling factor is. A conservative assumption is that smaller unrolling factors will minimize the probability of a data dependence between threads.

When a data dependence is violated and speculation fails, the corresponding *violation* is visible to software [3]—hence there are ways that unrolling can be altered depending on runtime performance: violation statistics can be fed back into the compiler through profile information, or dynamic recompilation techniques can be used to change the unrolling of a loop. Strip mining can be used to dynamically change the effective unrolling factor of a loop for profiling. This will not vary the time spent on synchronization, but will vary the memory dependence pattern and allow measurements of violation patterns. These techniques should be applied only to loops which do not yield to analysis.

Since we wish the following loop selection results to be independent of our unrolling results, in the subsequent sections we will use the *best* unrolling for each speculatively-parallel loop, together with whichever unrolling is independently best for each sequential loop.

8.3 Static Global Loop Selection

Given both the parallel and sequential performance for each loop, we can decide which loops are best to parallelize. Unfortunately, this decision is not always clear: two loops that perform well when parallelized can be nested as shown in Figure 8.6. We represent this relationship between loops by constructing a *loop graph* for the program as shown in Figure 8.7: each node in this graph is a loop in the original program. A directed edge indicates either nested loops, loops that are indirectly nested through procedure calls, or recursion.

8.3.1 Algorithm

At compile-time, we use the loop graph described above to decide which loops to parallelize. Recall from Section 8.2.1 that when we parallelize a loop, we surround it by a guarding `if` statement that prevents the execution of the parallel version of any nested loop—instead, the sequential version of any nested loop is executed. This way, if a loop contains a recursive call to itself then only the outer loop will create parallel threads. Hence we can simplify the loop graph by eliminating all back edges,[4] resulting in a directed, acyclic *loop graph*. Next we annotate each node in the loop graph with its *gain* (recall the definition of gain from Section 8.2.1); any negative gain is set to zero.

We want to identify the set of loops which will maximize total gain, and do so by applying the following gain-distribution procedure to each node in the order of a reverse depth-first-search:

[3] If violations are not directly visible to software, their frequency can instead be exposed through a performance counter

[4] In the case of mutual recursion, an arbitrary edge is chosen to break the cycle

```
void foo()
{
   bar();
   while(test1) {
      while(test2) {
         bar();
      }
   }
}
```

(a) Nested.

```
void bar()
{
   while(test3) {
   }
}
```

(b) Partially nested.

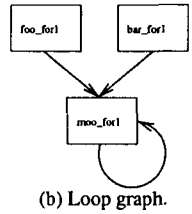
Figure 8.6: Nesting of loops may be complete or partial. The loop in `bar()` is only partially nested since it may be called either directly or from within the loops in `foo()`.

```
void foo()
{
   for(i = 0; i < 10; i++)
      printf("%d\n", moo(i));
}

void bar()
{
   for(i = 0; i < 20; i++)
      printf("%d\n", moo(i));
}

int moo(int i)
{
   sum = 1;
   for(j = 0; j < i; j++)
      sum = sum + moo(j);
   return sum;
}
```

(a) Source code.



(b) Loop graph.

Figure 8.7: Converting code into a loop graph.

1. Sum the gains of all child nodes.
2. Compare this sum to the gain for the current node.
3. If the current node's gain is greater, then remove all of the current node's outbound edges.
4. If the sum is greater than the current node's gain, then the current node's gain is set to the value of the sum.
5. If the gain is zero and there are no outbound edges, remove the current node from the graph.
6. Divide the gain for the current node equally among all parent nodes.

Upon completion of the gain-distribution procedure, the leaf nodes of the loop tree are the loops selected for parallelization. Note that this algorithm divides gain equally amongst all parent nodes, assuming that all paths are executed in equal frequency—path profiling information could be used to make this division of gain more accurate.

8.3.2 Results

We used the algorithm above to obtain the set of loops which maximize performance, transformed these loops for speculatively-parallel execution, and measured their performance. in Figure 8.8, the *O* bars plot the performance of this selection of loops.

This generates interesting results (10 of 25 benchmarks speed up[5]), but requires extensive profile information: for this selection we used all 1462 loop measurements. Collecting this amount of profile information in a production environment is impractical, so how else can we decide which loops to select for parallel execution?

[5]GO is an interesting exception: our profi le data assumes that each loop measurement is *independent* of all other loops, but when combined into a single program sometimes negative interactions can occur between parallelized loops.

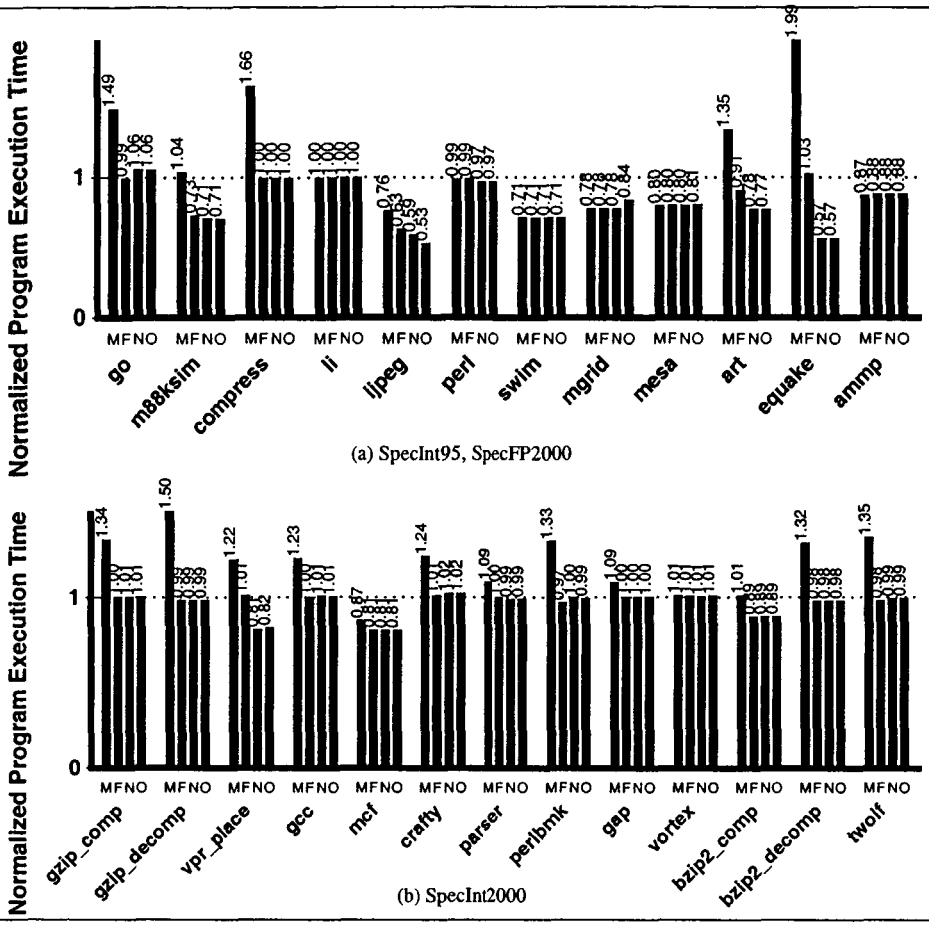


Figure 8.8: Program performance given different loop selection techniques. Performance is shown relative to an executable compiled with SUIF and with no parallel loops or instrumentation. *M* maximizes coverage; *F* maximizes coverage and then filters out loops which do not speed up; *N* filters out loops which do not speed up then maximizes coverage; and *O* maximizes speedup,

Coverage information (the fraction of execution time spent in a given region of code) is much easier to measure (eg., through basic block profiling), we tried re-running our loop selection algorithm maximizing *coverage* instead of gain, hoping that this would maximize the overall impact of TLS since we would be speculatively parallelizing the largest possible fraction of the program. The *M* bars in Figure 8.8 show the result of this maximal coverage approach. When these outermost loops in a program happen to be very parallel, TLS is quite effective. However, maximizing coverage can also maximize the overheads of TLS for loops with frequent dependences, resulting in slowdowns for 15 of the 25 benchmarks.

What is the difference between these two sets of loops? There are two reasons why the selection of any particular loop may be sub-optimal:

1. The loop may actually slow down when parallelized—it is better to just not parallelize such a loop.
2. The loop may be an outer loop which surrounds an inner loop that performs better—the larger threads of the outer loop may contain less parallelism than the smaller threads of the inner loop.

We try to measure the impact of each of these reasons separately.

Perhaps the most important reason why the *M* selection of loops perform poorly is because we include loops which really hinder performance. To quantify this claim we take the set of loops from the *M* experiment and filter out the loops which slow down when parallelized, as represented by the *F* bars in Figure 8.8. This should at least ensure that

programs never slow down under TLS, which is true with the exception of some minor slowdowns.[6] In addition, this filtering generates performance equal to (or better than) our "best" (*O*) results for 19 of the 25 benchmarks.

However, some applications still do not perform as well as the *O* experiment. Perhaps when we parallelize an outer loop we lose the opportunity to parallelize a more beneficial inner loop? In the *N* bars in Figure 8.8, we first filtered out loops which slowed down, and then maximized coverage on the remaining loops. For VPR_PLACE, M88KSIM, IJPEG, ART, and EQUAKE this has a significant positive impact, generating results that are close to our "best" results. This implies that a good selection algorithm may simply try to eliminate poorly performing loops, rather than attempting to find the "best" loops.

Can the compiler find these loops that perform poorly and filter them out? It can, but with extensive (and difficult to gather) profile information. We attempted to estimate the performace of loops using statistics that are more easily gathered, but we found that parallel performance is critically dependent on the actual run-time data dependence patterns between speculative threads. One of the main motivations for TLS is that data dependences in pointer rich codes are difficult to predict, and so it is not suprising that we are unable to predict them at compile time. In addition, the data dependence patterns for a program may depend on the input to that program, meaning that profile driven compilation may not produce good performance for inputs other than the one profiled.

The results of static region selection suggest that a compiler using a simple loop selection algorithm combined with run-time support for filter out poorly-performing loops can offer performance improvements equivalent to our best static global technique, without requiring extensive profiling. In the next section we shall develop and evaluate such techniques.

8.4 Dynamic Local Loop Selection

Taking advantage of information gathered at run-time allows us to find the best loops to parallelize without relying on complex compiler analysis, and also allows the system to adapt to changing inputs. In this section we evaluate a scheme where *all* loops are parallelized at compile time, but we decide which loops to execute in parallel at run time.

8.4.1 Code Transformation

Recall the guard code used in Section 8.2.1 to protect against nested parallelism and recursion. Similar code can be used to facilitate dynamic loop selection:

```
do{
   work();
} while(test);
```

is transformed to:

```
if(loop_start(loop_id)) {
   do_parallel{
      work();
   } while(test);
} else{
   do{
      work();
   } while(test);
}
loop_end(loop_id);
```

The `loop_start()` call returns `true` if the corresponding loop should be executed in parallel, and `false` otherwise. The `loop_end()` instruction notifies the run-time system when the corresponding loop ends, so that the time spent in the loop can be tracked. The `loop_end()` call is inserted at all loop exits. This mechanism allows the run-time system to measure the time spent executing a loop (eg., using a cycle counter), and also to decide whether a loop should be executed in parallel the next time it is encountered.

8.4.2 Impact of Transformation

It is important to quantify the overheads of transformation for dynamic selection. First, there is static overhead caused by code duplication, as reported in Figure 8.9. Second, there is dynamic overhead: in our prototype we add six

[6] `gcc` would not always produce identical code when we compiled a loop and instrumented it for profiling, and when we compiled the same loop into our benchmark. This difference caused minor measurement errors.

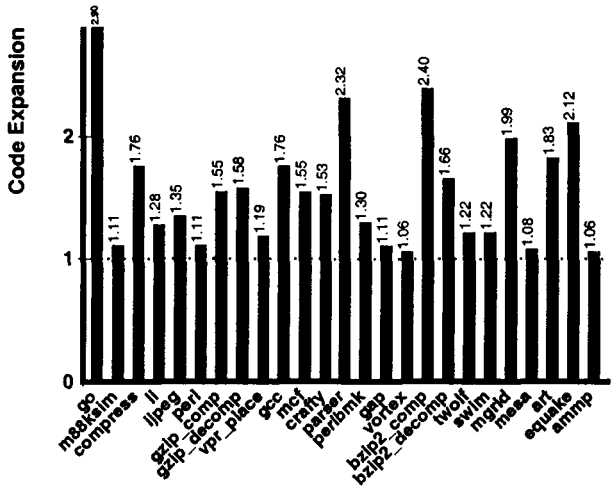


Figure 8.9: Impact of transformation on static code size.

dynamic instructions per loop for the `loop_start()` and `loop_end()` functions, including a branch. If we count the number of invocations of each loop per benchmark, multiply by six, and compare this to the dynamic instruction counts of our benchmarks we get the estimated overheads in Table 8.3.

Executing the dynamic program sequentially, we find that the overheads can be somewhat larger, as shown in Table 8.3 as well as the ***D*** bars of Figure 8.10. The extra overhead is due to the `gcc` compiler behaving conservatively in the presence of inline assembly—hence a more aggressive compiler back-end should yield even better performance than the results reported here.

Benchmark		Estimated Overhead (instr count)	Measured Overhead (cycles)
SpecInt95	go	4.57%	15.8%
	m88ksim	0.98%	0.8%
	compress	2.18%	9.4%
	li	4.75%	7.4%
	ijpeg	0.55%	2.4%
	perl	0.01%	-1.4%
SpecInt2000	gzip_comp	1.67%	4.4%
	gzip_decomp	6.05%	11.0%
	vpr_place	1.33%	4.4%
	gcc	2.11%	9.3%
	mcf	0.91%	3.0%
	crafty	1.45%	6.1%
	parser	2.90%	10.1%
	perlbmk	0.50%	-3.4%
	gap	0.40%	5.0%
	vortex	0.06%	1.4%
	bzip2_comp	2.88%	9.6%
	bzip2_decomp	2.16%	14.7%
	twolf	4.73%	9.0%
SpecFp2000	swim	0.00%	4.0%
	mgrid	0.07%	3.6%
	mesa	1.41%	11.8%
	art	1.31%	13.9%
	equake	9.05%	68.7%
	ammp	0.43%	2.7%

Table 8.3: Dynamic instruction overhead of dynamic loop selection transformation.

8.4.3 Run Time System

The run-time system is responsible for measuring the performance of each loop, and deciding whether to run it in parallel or not. Measurement is done using existing cycle counters in the CPU, while the run-time system can be

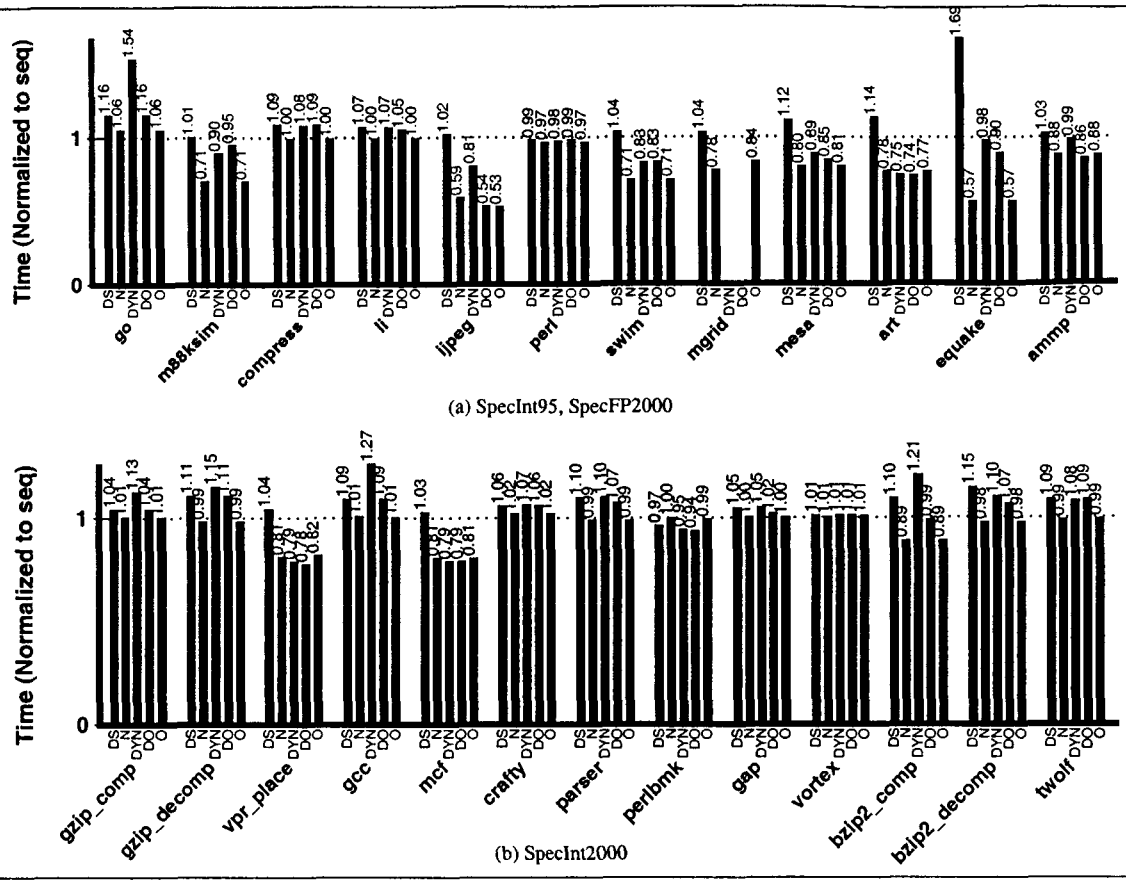


Figure 8.10: Program performance for dynamic loop selection. Performance is shown relative to an executable compiled with SUIF and with no parallel loops or instrumentation. ***DS*** shows a dynamic loop selection executable run sequentially — no loops are selected for parallel execution; ***N*** is same as the ***N*** bar in Figure 8.8, this the best we expect dynamic loop selection to perform (assuming no overhead); ***DYN*** runs each loop sequentially the first time it is encountered, on the second encounter runs it in parallel, then selects the best of the two for subsequent runs; ***DO*** shows the dynamic loop selection executable running the same regions as the ***O*** bar in Figure 8.8; ***O*** is the same as the ***O*** bar in Figure 8.8. *[Note: missing bars are due to not yet resolved simulator bugs. We apologize for the incomplete data.]*

implemented either in software or in hardware. The sequential performance of multiple loops can be gathered simultaneously, since nested sequential loops do not interact with each other. Unfortunately, only one nested parallel loop can be measured at a time to avoid interactions between parallel loops.

For simplicity, we assume that a measurement will be made over the entire execution of a loop: we measure it from the `loop_start()` call to the `loop_end()` call. This means that measuring a long-running loop executing in parallel may take significant time, during which no other parallel measurments can be made. Our first implementation measures the parallel execution of each loop when it is first encountered, and then measures its sequential execution (the next time it is encountered). Since *all* of our benchmarks have a single outer loop, the entire experiment only measured the parallel performance of the outermost loop for each benchmark, preventing any other loops from having their parallel execution measured. In hindsight, it is obvious why this did not perform well at all.

Our next approach is the opposite: we measure sequential execution of the first instance of a loop, and measure the parallel execution of the second instance that loop; this allows shorter-running inner loops to be evaluated more quickly (although the long-running outer loop is never fully evaluated). For a given loop, we collect both the sequential and parallel measurements and compare them to decide whether or not to execute future loop instances in parallel. In our results below we make this decision once for the remainder of execution—a more sophisticated implementation would periodically refresh the measurments to determine whether program behavior changes over time, or to see if the

behaviour of a loop is different for different invocations.

8.4.4 Results

The *DYN* bars in Figure 8.8 show the performance of dynamic loop selection using the "sample sequential first" approach outlined above. In spite of the overheads that dynamic loop selection has to overcome, 10 of the 25 benchmarks speed up. For comparison, the *DO* bars show the same executable run with the "best" set of loops from our static loop selection experiments. We see that in many cases (M88KSIM, SWIM,MESA,ART,VPR_PLACE,MCF and PERLBMK) the performance of dynamic loop selection is very close to that of static loop selection (once overheads are taken into account), indicating that dynamic loop selection has promise.

Unfortunately, there are also cases where dynamic loop selection hurts performance above and beyond the costs of overhead: GO, GZIP_COMP, GZIP_DECOMP, GCC and BZIP2_COMP are such cases. The cost of sampling poorly-performing loops can degrade performance significantly, which motivates having a mechanism to abort a parallel sample if performance is particularly poor. The measurements made at runtime could also be fed back into a dynamic compilation system to remove particularly poor loops from consideration, reducing the observed overheads.

These experiments show the potential for loop selection at run-time, but when the decision for a given loop is only made once: in these measurements once a loop has been sampled both in parallel and sequentially then no further measurement is done. We have performed preliminary experiments which demonstrate that the behavior of certain performance-critical loops varies over the execution of the program. Therefore a more active sampling technique could detect changes in program behavior and dynamically adapt the set of loops selected for parallel execution.

8.5 Summary

When we exploit TLS to automatically carve up a sequential program into parallel threads, the choice of *where* to demarcate thread boundaries is crucial to achieving good performance. Within the context of using loops as our source of parallelism (which represent over 97% of execution time for 23 out of the 25 applications we studied), there are two important questions to answer: (i) *which* loops should be parallelized, and (ii) *how* should the iterations within those loops be grouped together (via unrolling) to form thread boundaries? Ideally there would be a simple answer to these questions, such as *"always parallelize innermost loops and always unroll them by a factor of four."* Given that the original motivation for TLS was the unpredictable nature of data dependences in integer programs, however, it should come as no surprise that the answers to these questions are far more complex, and are difficult to evaluate statically at compilation time.

Regarding loop unrolling, we observe the following. First, a surprisingly large fraction of loops in *integer* (as opposed to *floating-point*) applications have trip counts that are so small (e.g., two or three iterations) that unrolling them would also eliminate them as a source of parallelism. Given sufficiently large trip counts, the success of loop unrolling depends upon how it affects: (i) *overhead*, (ii) *synchronization stalls*, and (iii) *failed speculation*. Of these three factors, the impact on overhead is easiest to predict: iterations with fewer than 30 instructions will only benefit from TLS if they are aggressively unrolled to reduce overheads, and iterations with over 300 instructions are already large enough that overheads are not a concern.

The impact of unrolling on synchronization stalls and failed speculation is mixed: each component can become either significantly better or significantly worse, and predicting these behaviors *a priori* (i.e. without simply unrolling the code and observing what impact that has) is extremely difficult. Regarding synchronization stalls, the good news is that we can reason about this effect at compilation time. By observing the size of the *critical forwarding path* [84] relative to the thread size, the compiler can make a rough approximation of whether synchronization stalls are likely to be a problem; if so, it can internally "simulate" the impact of unrolling by running the analysis phase of the scheduling algorithm [84] on the loop body for various unrolling factors, and emit unrolled code only if it favorably reduces the critical forwarding path. Although this brute-force approach may increase compilation time, at least it can be dealt with in the compiler. In contrast, it is unclear how to predict the impact of unrolling on failed speculation without simply observing this effect at execution time. The only good news is that we can use strip-mining to generate a single static excutable for the loop that will allow us to measure the impact of arbitrary unrolling factors on failed speculation; hence collecting this profiling information will require relatively little execution overhead.

Regarding the choice of *which* loops to parallelize, we observe that traditional control-flow profiling information is not sufficient to make a good choice. Instead, we require accurate measurement of the performance of both parallel and sequential executions of each loop (perhaps done using the on-chip cycle counters that are now available on most

processors). With this information in hand, execution time can be reduced by up to 44% for integer benchmarks and up to 54% for floating point benchmarks. Since such information is difficult to provide to the compiler, we explore techniques for making the thread selection decision dynamically at run time. Although the exploratory loop measurement process can be costly, 9 out of 25 benchmarks speed up when using dynamic loop selection.

Bibliography

[1] W. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of ISCA 27*, June 2000.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[3] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *MICRO-31*, December 1998.

[4] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiling. In *Proc. ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, 1998.

[5] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of ISCA 27*, June 2000.

[6] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of Micro-29*, 1996.

[7] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, Mass., 1988.

[8] Anasua Bhowmik and Manoj Franklin. A fast approximate interprocedural analysis for speculative multithreading compiler. In *17th Annual ACM International Conference on Supercomputing*, 2003.

[9] A. Brown, T. C. Mowry, and O. Krieger. Compiler-Based I/O Prefetching for Out-of-Core Applications. *ACM Transactions on Computer Systems*, 19(2):111–170, May 2001.

[10] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[11] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *International Symposium on Computer Architecture*, 1999.

[12] P. P Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. *Three Superblock Scheduling Models for Superscalar and Superpipelined Processors*. Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, 1991.

[13] D. K. Chen and P. C. Yew. Statement re-ordering for DOACROSS loops. In *International Conference on Parallel Processing*, pages 24–28, August 1994.

[14] D. K. Chen and P. C. Yew. Redundant synchronization elimination for doacross loops. *IEEE Transactions on Parallel and Distributed System*, 10(5):459–470, 1999.

[15] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, 2003.

[16] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C.Yew. Data dependence profiling for speculative optimization. In *13th International Conference on Compiler Construction*, Barcelona, Spain, March 2004.

[17] G. Chrysos and J. Emer. Memory dependency prediction using store sets. June 1998.

[18] George Chrysos and Joel Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th ISCA*, June 1998.

[19] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of ISCA 27*, June 2000.

[20] M. Cintra and J. Torrellas. Learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA02*, 2002.

[21] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th HPCA*, Feb 2002.

[22] Broadcom Corporation. The Sibyte SB-1250 Processor. http://www.sibyte.com/mercurian.

[23] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. Technical report. http://www.spechbench.org.

[24] Standard Performance Evaluation Corporation. The SPEC2000 Benchmark Suite. Technical report. http://www.spechbench.org.

[25] Standard Performance Evaluation Corporation. The SPEC95 Benchmark Suite. Technical report. http://www.spechbench.org.

[26] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, 1986.

[27] R. Cytron. Doarcoss: Beyond vectorization for multiprocessors. In *Int'l. Conf. on Parallel Processing*, August 1986.

[28] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors.

[29] J. Emer. Ev8: The post-ultimate alpha.(keynote address). In *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[30] Brian A. Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *ISCA 2001*, 2001.

[31] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 13, June 1981.

[32] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin – Madison, 1993.

[33] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[34] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report EE Department TR #1080, Technion–Israel Institute of Technology, 1996.

[35] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of the 6th ASPLOS*, pages 183–195, October 1994.

[36] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[37] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[38] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. Technical Report 1334, Computer Sciences Department, University of Wisconsin-Madison, July 1997.

[39] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98*, November 1998.

[40] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of ASPLOS-VIII*, October 1998.

[41] L. Howard Holley and Barry k. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, 7(1), January 1981.

[42] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99*, October 1999.

[43] Jens Knoop and Oliver Ruthing. Lazy code motion. In *Proc. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 92.

[44] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[45] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, 1996.

[46] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, September 2003.

[47] G. Tyson M. Farrens and A.R. Pleszkun. A study of single-chip processor/ cache organizations for large number of transistors. In *Proceedings of ISCA 21*, pages pp. 338–347, 1994.

[48] P. Marcuello and A. Gonzlez. Clustered Speculative Multithreaded Processors. In *Proc. of the ACM Int. Conf. on Supercomputing*, June 1999.

[49] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *International Symposium on Microarchitecture*, November 1999.

[50] Pedro Marcuello and Antonio González. Thread-Spawning Scheme for Speculative Multithreading. In *Proceedings of the 8th HPCA*, February 2002.

[51] Pedro Marcuello, Jordi Tubella, and Antonio Gonzsslez. Value prediction for speculative multithreaded architectures. In *Proceedings of Micro-32*, Haifa, Israel, November 1999.

[52] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, 1987.

[53] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *SIGSOFT workshop on on Program analysis for software tools and engineering Snowbird*, June 2001.

[54] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. June 1997.

[55] Andreas I. Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th ISCA*, June 1997.

[56] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[57] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27, pages 62–73, October 1992.

[58] A. Nicolau. Run-time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, 38:663–678, May 1989.

[59] J. Oplinger, D. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[60] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computing*, September 1980.

[61] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-1996-1328, University of Wisconsin-Madison, 1996.

[62] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed System*, 10(2):160–172, 1999.

[63] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of Micro 30*, 1997.

[64] Y. Sazeides and J. E. Smith. The Predictability of Data Values. *Proceedings of Micro 13*, pages 248–258, December 1997.

[65] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA 22*, pages 414–425, June 1995.

[66] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[67] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of ISCA 27*, June 2000.

[68] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the 8th HPCA*, February 2002.

[69] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving Value Communication For Thread-Level Speculation. In *Proceedings of the 8th HPCA*, February 2002.

[70] Z. Sura, C.-L. Wong, X. Fang, J. Lee S.P. Midkiff, and D. Padua. Automatic implementation of programming language consistency models. In *Sixth International Symposium on Parallel Architectures, Algorithms and Networks (LCPC'02)*, 2002.

[71] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[72] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *HotChips '99*, August 1999.

[73] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, 48(9), September 1999.

[74] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA 22*, pages 392–403, June 1995.

[75] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA 1996*, May.

[76] T. N. Vijaykumar and Gurindar S. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of Micro-31*, December 1998.

[77] T.N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, January 1998.

[78] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, 1997.

[79] Chih-Po Wen and Katherine A. Yelick. Compiling Sequential Programs for Speculative Parallelism. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPDS)*, December 1993.

[80] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proc. ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[81] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Massachusetts, 1989.

[82] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.

[83] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of ASPLOS-X*, October 2002.

[84] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *Proceedings of the 10th ASPLOS*, Oct 2002.

[85] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimizations to Accelerate Scalar Value Communication Between Speculative Threads. Technical Report CMU-CS-02-162, School of Computer Science, Carnegie Mellon University, August 2002.

[86] A. Zhai, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *The 2nd International Symposium on Code Generation and Optimization*, March 2004.

[87] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, June 1987.

[88] C. B. Zilles and G. S. Sohi. Master/Slave Speculative Parallelization with Distilled Programs. Technical Report TR-1438, Computer Sciences Department, University of Wisconsin-Madison, April 2002.