Center for Reliable and High-Performance Computing
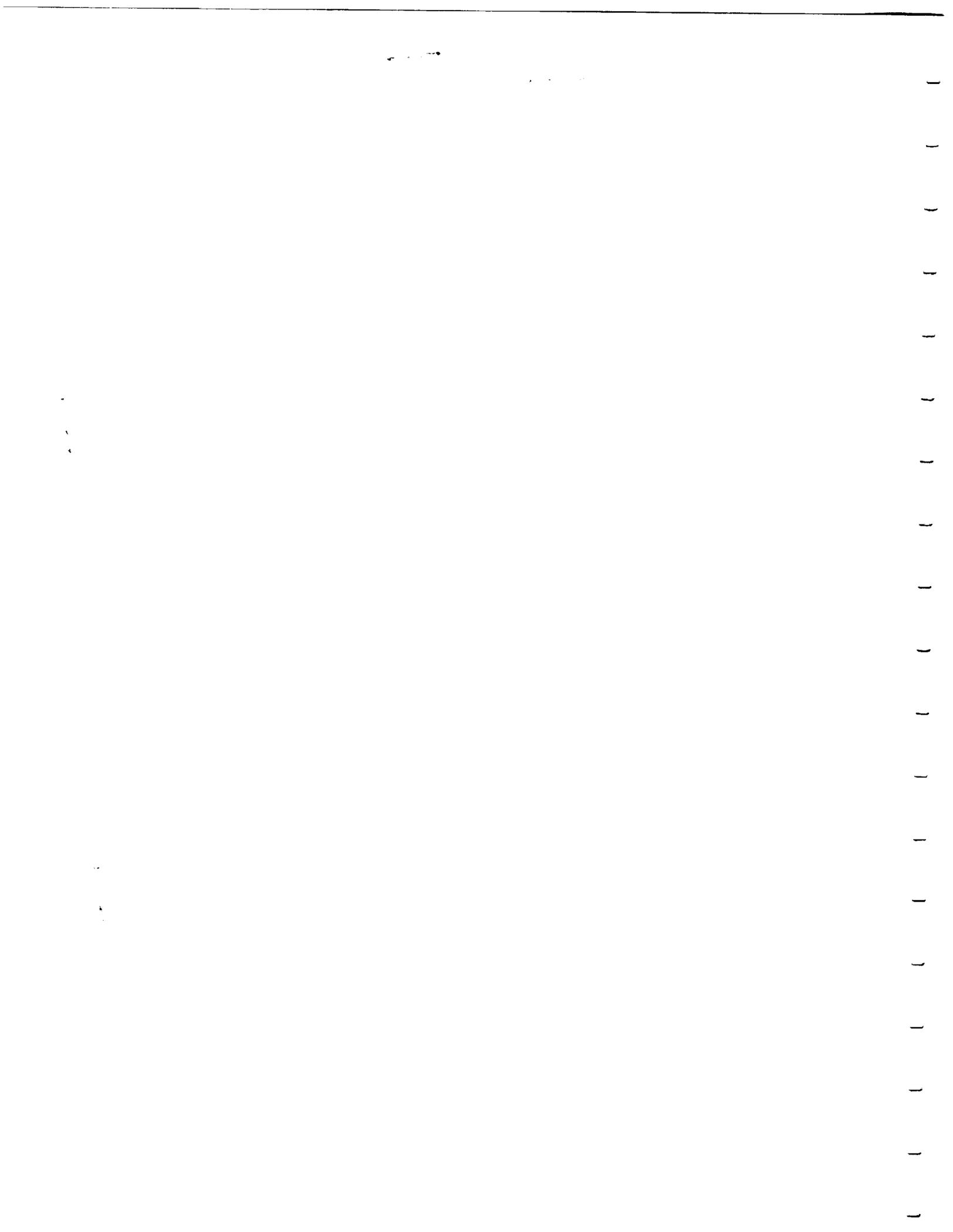
THE EFFECT OF CODE EXPANDING OPTIMIZATIONS ON INSTRUCTION CACHE DESIGN

William Y. Chen
Pohua P. Chang
Thomas M. Conte
Wen-mei W. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
Unclassified	None

2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
UILU-ENG-91-2227 CRHC-91-17	

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
Coordinated Science Lab University of Illinois	N/A	NCR, NSF, AMD, NASA

6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code)
1101 W. Springfield Avenue Urbana, IL 61801	Dayton, OH 45420 Washington DC 20550 Langley VA 20200

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
7a		N00014-91-J-1283 NASA NAG 1-613

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
7b				

11. TITLE (Include Security Classification)

The Effect of Code Expanding Optimizations on Instruction Cache Design

12. PERSONAL AUTHOR(S) Chen, William, Pohua Chang, Thomas Conte and Wen-Mei Hwu

13a. TYPE OF REPORT	13b. TIME COVERED		14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____	TO _____	91-05-23	37

16. SUPPLEMENTARY NOTATION

17	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	instruction placement, inline expansion, superscalar optimizations, cache design, cache memory, C compiler, load-forwarding

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper shows that code expanding optimizations have strong and non-intuitive implications on instruction cache design. Three types of code expanding optimizations are studied in this paper: instruction placement, function inline expansion, and superscalar optimizations. Overall, instruction placement reduces the miss ratio of small caches. Function inline expansion improves the performance for small cache sizes, but degrades the performance of medium caches. Superscalar optimizations increases the cache size required for a given miss ratio. On the other hand, they also increase the sequentiality of instruction access so that a simple load-forward scheme effectively cancels the negative effects. Overall, we show that with load forwarding, the three types of code expanding optimizations jointly improve the performance of small caches and have little effect on large caches.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

The Effect of Code Expanding Optimizations on Instruction Cache Design

William Y. Chen   Pohua P. Chang   Thomas M. Conte   Wen-mei W. Hwu *

April 29, 1991

Abstract

This paper shows that code expanding optimizations have strong and non-intuitive implications on instruction cache design. Three types of code expanding optimizations are studied in this paper: instruction placement, function inline expansion, and superscalar optimizations. Overall, instruction placement reduces the miss ratio of small caches. Function inline expansion improves the performance for small cache sizes, but degrades the performance of medium caches. Superscalar optimizations increases the cache size required for a given miss ratio. On the other hand, they also increase the sequentiality of instruction access so that a simple load-forward scheme effectively cancels the negative effects. Overall, we show that with load forwarding, the three types of code expanding optimizations jointly improve the performance of small caches and have little effect on large caches.

Index terms - C compiler, code optimization, cache memory, code expansion, load forwarding, instruction placement, function inline expansion, superscalar optimizations.

*The authors are with the Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, Illinois, 61801.

1

1 Introduction

Compiler technology plays an important role in enhancing the performance of processors. Many code optimizations are incorporated into a compiler to produce code that is comparable or better than hand-written machine code. Classic code optimizations decrease the number of executed instructions [1]. However, there are factors limiting the effectiveness of these optimizations. For example, small function bodies limit the scope of optimization and scheduling. To increase the scope of code optimization, inline function expansion is performed by many compilers [2] [3] [4]. Function inlining replaces a function call with the function body. To further enlarge the scope of code optimization and scheduling, compilers unroll loops by duplicating the loop body several times. The IMPACT-I C compiler utilizes inline expansion, loop unrolling, and other code optimization techniques. These techniques increase the execution efficiency at the cost of increasing the overall code size. Therefore, these compiler optimizations can affect the instruction cache performance.

This paper examines the effect of these code expanding optimizations on the performance of a wide range of instruction cache configurations. The experimental data indicate that code expanding optimizations have strong and non-intuitive implications on instruction cache design. For small cache sizes, the overall cache miss ratio of the expanded code is lower than that of the code without expansion. The opposite is true for large cache sizes. This paper studies three types of code expanding optimizations: instruction placement, function inline expansion, and superscalar optimizations. Overall, instruction placement increases the performance of small caches. Function inline expansion improves the performance of small caches, but degrades that of medium caches. Superscalar optimizations increases the cache size required for a given miss ratio. However, they also increase the sequentiality of instruction access so that a simple load-forward scheme removes

the performance degradation. Overall, it is shown that with load forwarding, the three types of code expanding optimizations jointly improve the performance of small caches and have little effect on large caches.

1.1 Related Work

Cache memory is a popular and familiar concept. Smith studied cache design tradeoffs extensively with trace driven simulations [5]. In his work, many aspects of the design alternatives that can affect the cache performance were measured. Later, both Smith and Hill focused on specific cache designs parameters. Smith studied the cache block (line) size design and its effect on a range of machine architectures, and found that the miss ratios for different block sizes can be predicted regardless of the workload used [6]. The causes of cache misses were categorized by Hill and Smith into three types: conflict misses, capacity misses, and compulsory misses [7]. The loop model was introduced by Smith and Goodman to study the effect of replacement policies and cache organizations [8]. They showed that under some circumstances, a small direct mapped cache performs better than the same cache using fully associativity with LRU replacement policy. The tradeoffs between a variety of cache types and on-chip registers were reported by Eickemeyer and Patel [9]. This work showed that when the chip area is limited, a small- or medium-sized instruction cache is the most cost effective way of improving processor performance. Przybylski *et al.* studied the interaction of cache size, block size, and associativity with respect to the CPU cycle time and the main memory speed [10]. This work found that cache size and cycle time are dependent design parameters. Alpert and Flynn introduced an utilization model to evaluate the effect of the block size on cache performance [11]. They considered the actual physical area of caches and found that larger block sizes have better cost-performance ratio. All of these studies assumed an invariant

compiler technology and did not consider the effects of compiler optimizations on the instruction cache performance.

Load forwarding is used to reduce the penalty of a cache miss by overlapping the cache repair with the instruction fetch. Hill and Smith evaluated the effects of load forwarding for different cache configurations [12]. They concluded that load forwarding in combination with prefetching and sub-blocking increases the performance of caches. In this paper a simpler version of the load-forward scheme is used, where neither prefetching nor sub-blocking is performed. The effectiveness of this load-forward technique is measured by comparing the cache performance of code without optimizations and with code expanding optimizations. Load forwarding potentially can hide the effects of code expanding optimizations.

Davidson and Vaughan compared the cache performances of three architectures with different instruction set complexities [13]. They have shown that less dense instruction sets consistently generate more memory traffic. The effect of instruction sets of over 50 architectures on cache performance has been characterized by Mitchell and Flynn [14]. They showed that intermediate cache sizes are not suited for less dense architectures. Steenkiste [15] was concerned with the relationship between the code density pertaining to instruction encoding and instruction cache performance. He presented a method to predict the performance of different architectures based on the miss rate of one architecture. Unlike less dense instruction sets which typically have higher miss rate for small caches [13], we show that code expansion due to optimizations improves performance of small caches, and degrades that of large caches. Our approach is also different from these previous studies in that the instruction set is kept constant. A load/store RISC instruction set whose code density is close to that of the MIPS R2000 instruction set is assumed.

Cnderman and Flynn have simulated the effects of classic code optimizations on architecture

design decisions [16]. Classic code optimizations do not significantly alter the actual working sets of programs. In contrast, in this paper, classic code optimizations are always performed; code expanding optimizations that enlarge the working sets are the major concern. Code expanding optimizations increase the actual code size and change the instruction sequential and spatial localities.

1.2 Outline Of This Paper

Section 2 describes the instruction cache design parameters and the performance metrics. The cache performance is explained using the recurrence/conflict model [17]. Section 3 describes the code expanding optimizations and their effects on the target code and the cache design. Section 4 presents and analyzes experimental results. Section 5 provides some concluding remarks.

2   Instruction Cache Design Parameters

2.1   Performance Metrics with Recurrences and Conflicts

The dimension of a cache is expressed by three parameters: the cache size, the block size, and the associativity of the cache [5]. The size of the cache, 2^C, is defined by the number of bytes that can simultaneously reside in the cache memory. The cache is divided into b blocks, and the block size, 2^B, is the cache size divided by b. The associativity of a cache is the number of cache blocks that share the same cache set. An associativity of one is commonly called a direct mapped cache, and an associativity of 2^{C-B} defines a fully associative cache.

The metric used in many cache memory system studies is the cache miss ratio. This is the ratio of the number of references that are not satisfied by a cache at a level of the memory system

hierarchy over the total number of references made at that cache level. The miss ratio has served as a good metric for memory systems since it is characteristic of the workload (e.g., the memory trace) yet independent of the access time of the memory elements. Therefore, a given miss ratio can be used to decide whether a potential memory element technology will meet the required bandwidth for the memory system.

The recurrence/conflict model [17] of the miss ratio will be used to analyze the cause of cache misses. Consider the trace in Figure 1. a_1, a_2, a_3, and a_4 are the first occurrence of an access, and they are *unique* in the trace. The *recurrences* in the trace are accesses a_5, a_6, a_7 and a_8. Without a context switch, all these four recurrences would result in a hit in an infinite cache. In the ideal case of an infinite cache and in the absence of context-switching, the *intrinsic* miss ratio is expressed as,
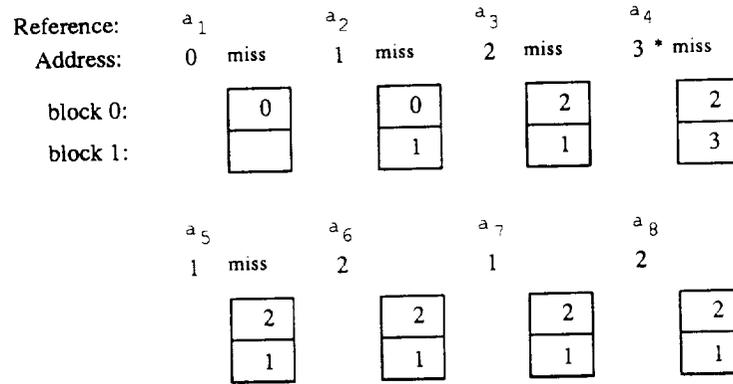
$$\rho_o = \frac{N - R}{N}.\qquad(1)$$

where R is the total number of recurrences and N is the total number of references. Note that an access can be of only two types: either a *unique* or a *recurrent* access. Non-ideal behavior occurs due to *conflicts*, and this paper considers only the *dimensional conflicts*; multiprogramming conflicts are considered in [18].

A *dimensional conflict* is defined as an event which converts a recurrent access into a miss due to limited cache capacity or mapping inflexibility. For illustration, consider a direct mapped cache composed of two one-byte blocks as shown in Figure 2. A miss occurs for recurrent access a_5

Reference	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
Address	0	1	2	3	1	2	1	2

Figure 1: An example trace of addresses.

Reference: a_1 a_2 a_3 a_4
Address: 0 miss   1 miss   2 miss   3 * miss

block 0: | 0 | | 0 | | 2 | | 2 |
block 1: | | | 1 | | 1 | | 3 |

            a_5 a_6 a_7 a_8
            1 miss   2   1   2

              | 2 | | 2 | | 2 | | 2 |
              | 1 | | 1 | | 1 | | 1 |

* Dimensional conflict

Figure 2: An example two-block direct-mapped cache behavior.

because reference a_4 purges address 1 from the cache due to insufficient cache capacity. Hence, a_4 represents a dimensional conflict for the recurrence a_5. The other misses, a_1, a_2, a_3 and a_4, occur because these are the first references to addresses 0, 1, 2 and 3, respectively (i.e., they are *unique accesses*). Therefore, the following formula can be used for deriving the cache miss ratio, ρ, for a given trace, and a given cache dimension:

$$\rho = \frac{N - (R - C_D)}{N} = \rho_o + \frac{C_D}{N}. \tag{2}$$

where C_D is the total number of dimensional conflicts, and ρ_o is the intrinsic miss ratio.

In a simple design, when a cache miss occurs, instruction fetch stalls and the instruction cache waits for the appropriate cache block to be filled. After instruction cache repair is completed, the instruction fetch resumes. The number of stalled cycles is determined by three parameters: the initial cache repair latency (L), the block size, and the cache-memory bandwidth (β). For a single cache miss, the number of stalled cycles is the initial cache repair latency plus the number of transfers required to repair the cache block. The total miss penalty without load forwarding, t_n, is expressed by the number of total misses multiplied by the number of stalled cycles for a single

cache miss.

$$t_n = (N - (R - C_D)) \times (L + \frac{2^B}{\beta}).$$ (3)

This is the miss-penalty model used when load forwarding is not assumed. The miss penalty ratio is calculated by dividing the miss penalty, t_n, by N.

2.2 Load Forwarding

Load forwarding was evaluated by Hill and Smith [12]. They concluded that load forwarding in combination with prefetching and sub-blocking increases the performance of the cache. In this paper, we use a simpler version of the load forwarding scheme where neither prefetching nor sub-blocking is performed. The state transition diagram for load forwarding is shown in Figure 3. The instruction cache is in the standby state initially (state 0). When a cache miss occurs, the instruction fetch stalls (state 1). Instead of waiting for the entire cache block to be filled before resuming, the cache loads the block from the currently-referenced instruction and forwards the instruction to the instruction fetch unit (state 2). Furthermore, if the instruction reference stream is sequential, each subsequent instruction is forwarded to the instruction fetch unit until the end of the block is reached or a taken branch is encountered. Any remaining unfilled cache-block bytes are repaired in the normal manner, and the instruction fetch stalls (state 3). This load forwarding scheme requires no sub-block valid bits and therefore has a simpler logic for cache block repair than sub-block-based schemes.

An example of the cache-block repair process with load forwarding is provided in Figure 4. Reference X results in a miss. It takes L cycles before this reference is placed in the appropriate block location and is forwarded to the fetch unit. Reference Y is a sequential access, thus it is considered as a hit. It is placed in the cache and forwarded to the fetch unit. Reference Z breaks
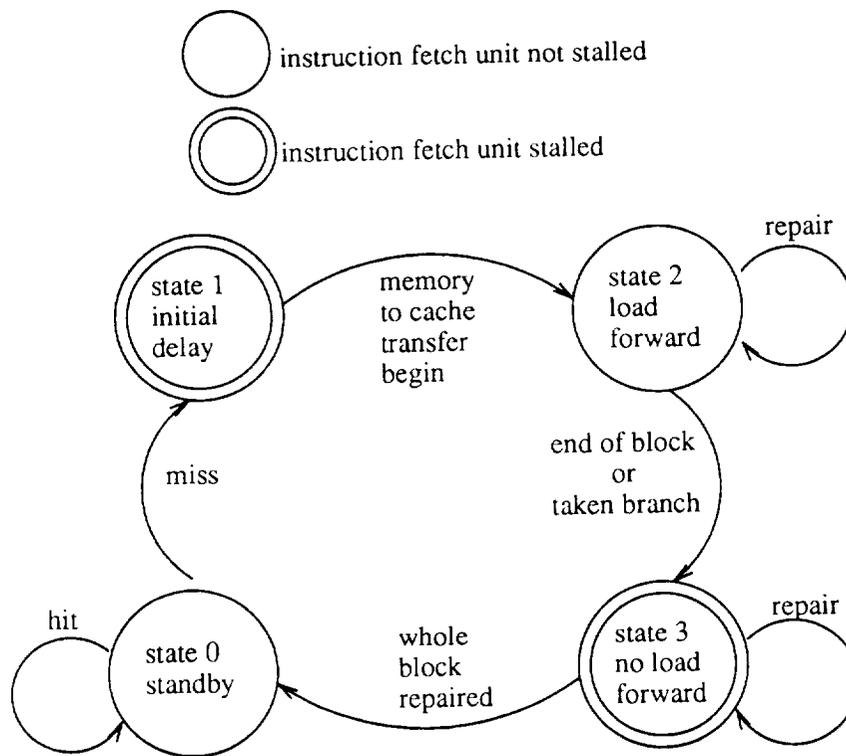
Figure 3: State transition diagram of the load forwarding process.

Cycle:	0	L	L+1
Status:	stall and repair	forward	forward
Reference:	X	X	Y
Address:	1 miss	1	2 hit

block 0:
block 1:

Cycle:	L+2	L+3	2*L+3
Status:	stall and repair	stall and repair	forward
Reference:	Z	Z	Z
Address:	4 miss	4	4

block 0:
block 1:

Figure 4: An example of the load forwarding process.

the sequential-reference stream, load forwarding stops, and cache repair of block 0 continues. At cycle $L+2$, the end of the block is reached, and the cache repair continues from the beginning of the cache block. At cycle $L+3$, the entire cache block is filled, the fetch unit continues with the next instruction reference. The block wrap around time is assumed to be negligible compared to the total block-repair time [1]. References X and Y are sequential and constitute a *run length* (the number of sequential instructions before a taken branch) of 2.

For the ith cache miss, if the total number of bytes where the instruction fetch and cache repair

---

[1] For the actual hardware implementation, the cache repair can start at the beginning of the cache block. When the location of the instruction to be fetched is encountered within the cache block, load forwarding begins. Load forwarding terminates when the end of the block is reached or when a taken branch is encountered. Cache repair stops at the end of the block. The miss penalty incurred by this method is the same as the one presented in the paper.

overlap is represented by $S[i]$, the total miss penalty with load forwarding, t_l, is expressed as

$$t_l = t_n - t_S \tag{4}$$

where t_S is

$$t_S = \sum_{i=1}^{(N-R)+C_D} \frac{S[i]}{\beta}. \tag{5}$$

t_S measures the number of cycles saved by load forwarding. Equation 4 is the miss-penalty model used when load forwarding is assumed. The miss penalty ratio with load forwarding is calculated by dividing the miss penalty, t_l, by N.

The saved cycles expressed in Equation 5 is constrained by two factors. First, load forwarding is limited by the sequentiality of the instruction reference stream. The more sequential the instruction reference stream is, the more overlap between the cache repair and load forwarding cycles that can be achieved. Second, assuming the sequentiality of the referencing stream is not a problem, load forwarding is performed only from the missed reference until the end of the block. Thus the savings is highly dependent upon the location of the miss within the cache block. The sequentiality of the reference stream can be increased by appropriate compiler optimizations and this will be discussed in Section 3. This second factor is highly variable and dependent upon the instruction reference stream and the block size.

3 Optimizations and Code Transformations

3.1 Base Optimizations

A standard set of classic optimizations is available in commercial compilers today (see Table 1). The goal of these optimizations is to reduce the execution time. Local optimizations are performed

11

Local	Global
constant propagation	constant propagation
copy propagation	copy propagation
common subexpression elimination	common subexpression elimination
redundant load elimination	redundant load elimination
redundant store elimination	redundant store elimination
constant folding	dead code removal
strength reduction	loop invariant code removal
constant combining	loop induction strength reduction
operation folding	loop induction elimination
operation cancellation	global variable migration
dead code removal	loop unrolling
code reordering	

Table 1: Base optimizations.

within basic blocks, whereas global optimizations are performed across operations in different basic blocks. In this paper, these classic code optimizations are always performed on the compiled programs.

3.2 Execution Profiler

Execution profiling is performed on all measured benchmarks. The IMPACT-I profiler translates each target C program into an equivalent C program with additional probes. When the equivalent C program is executed, these probes record the basic block weights and the branch characteristics for each basic block. Profile information is used to guide the code expanding optimizations. The profile information is collected using an average 20 program inputs per benchmark. An additional input is then used to measure the cache performance.

3.3 Instruction Placement

Reordering program structure to improve the memory system performance is not a new subject. In more recent literature regarding instruction caches, instruction placement has been shown to improve performance [19] [20] [21]. The IMPACT-I C compiler instruction placement algorithm improves the efficiency of caching in the instruction memory hierarchy [19]. Based on dynamic profiling, this algorithm increases the sequential and spatial localities, and decreases cache mapping conflicts of the instruction accesses.

For a given function body, several steps are taken to reorder the instruction sequence. For each function, basic blocks which tend to execute in sequence are grouped into traces [22] [23]. Traces are the basic units used for instruction placement. The algorithm starts with the function entrance trace and expands the placement by placing the most important descendent after it. The placement continues until all the traces with non-zero execution profile count have been placed. Traces with zero execution count are moved to the bottom of the function, resulting in a smaller effective function body.

Reordering the basic blocks does not increase the program size significantly. The overall sequentiality of the resulting code is increased (i.e. the number of taken branches are reduced) due to the formation of traces, and this may increase the need for a larger cache block size. For the same cache size, an increase in block size translates to a decrease in tag store. The overall locality of the resulting code is increased due to the placement of more important traces at the beginning of the function.

3.4 Function Inline Expansion

Function inline expansion replaces the frequently invoked function calls with the function body. The importance of inline expansion as an essential part of an optimizing compiler has been described by Allen and Johnson [24]. Several optimizing compilers perform inline expansion. For example, the IBM PL.8 compiler does inline expansion of all leaf-level procedures [25]. In the GNU C compiler, the programmer can use the keyword *inline* as a hint to the compiler for inline expanding function calls [2]. The Stanford MIPS C compiler examines the code structure (e.g., loops) to choose the function calls for inline expansion [26]. The IMPACT-I C compiler has an algorithm that automatically performs inter-file inlining assisted by the profile information where only the important function call sites are considered [4]. Inlining is done primarily to enlarge the scope of optimization and scheduling.

Since the callee is expanded into the caller, inline expansion increases the spatial locality and decreases the number of function calls. This transformation increases the number of unique references, which may result in more misses. However, a decrease in the miss ratio may also occur, because without inline expansion the callee has the potential to replace the caller in the instruction cache. With inline expansion, this effect is reduced. Inline expansion provides large functions to enlarge the size of traces selected. This enlargement of function bodies helps to further the effectiveness of instruction placement. With an increase in the sequentiality of the referencing stream, an improvement in the performance of load forwarding can be expected.

3.5 Optimizations for Superscalar Processors

Since basic blocks typically contain few instructions, there is little parallelism within a basic block. For superscalar processors, many code transformations are necessary in order to increase the num-

ber of instructions available for scheduling. Many researchers have shown the effectiveness of these optimizations [27] [28] [29]. Although these optimizations are frequently used for super-scalar processors, these optimizations are also useful for scalar processors (e.g., MIPS C compiler performs automatic loop unrolling [3]). The following superscalar optimizations have been implemented in the IMPACT-I C compiler and are performed in addition to function inline expansion and instruction placement. They have been shown to provide significant speedup on superscalar processors [30].

Super-block formation: A super-block is a sequence of instructions that can be reached only from the top instruction and may contain multiple branch instructions. A trace can be converted to a super-block by creating a copy of the trace and by redirecting all control transfers to the middle of the trace to the duplicate copy; thus, super-block formation, or trace duplication, increases code optimization and scheduling freedom.

Loop unrolling: The body of a loop is duplicated to increase the number of instructions in the super-block. To unroll the loop N times, the body of the loop is duplicated (N - 1) times. For multiple instruction issue processors, the IMPACT-I C compiler typically unrolls small loops four or more times. For larger loops, N decreases according to the loop size.

Loop peeling: Many loops iterate very few times, (e.g., less than ten). For these loops, loop unrolling and software pipelining are less effective because the execution time spent in the parallel section (the optimized loop body) is not substantially longer than in the sequential section (the loop prologue and epilogue). An alternative approach to loop unrolling is to peel off enough iterations, such that the loop typically executes as a straight-line code.

Branch target expansion: Instruction placement and super-block formation introduce many branch instructions. Branch target expansion helps to eliminate the number of taken branches by

program	description	object code size (bytes)	instruction references
cccp	GNU C preprocessor	20400	2.89×10^7
eqntott	truth table generator	15256	1.47×10^8
espresso	boolean minimization	61264	5.48×10^7
mpla	pla layout	138808	1.07×10^8
tbl	format table for troff	24804	3.08×10^7
xlisp	lisp interpreter	31920	1.46×10^8
yacc	parsing program generator	21320	3.47×10^7

Table 2: Benchmark program characteristics.

copying the target basic block of a frequently taken branch into its fall-through path. The number of static instructions increases due to this optimization.

Super-block formation, loop unrolling, loop peeling, and branch target expansion increase the sequentiality of the code. Loop unrolling and loop peeling decrease both spatial and temporal locality. A reduction in cache performance can be expected due to a decrease in spatial locality. The increased code size and increased unique references can be expected to increase the cache size requirement.

4 Experiments and Analysis

4.1 Benchmark Programs

Table 2 shows the benchmark programs that are used in this paper. Three of the programs, *eqntott*, *espresso*, and *xlisp*, are from the SPEC[2] benchmark set [31]. Four other C programs, *mpla*, *cccp*, *yacc*, and *tbl*, are commonly used scalar programs. The *object code size* column gives the program size in bytes without any code expanding optimizations. The size of these benchmark

---

[2] University of Illinois is a member of SPEC.

programs are large enough for studying instruction caches. The *instruction references* column gives

the corresponding number of dynamic instruction references. These instruction references are for

the full run of each benchmark program, no sampling or reference partitioning is used.

4.2 Measurement Tools

The measurement results are generated by trace driven simulation. To collect the instruction

traces, the compiler's code generator was modified to insert probes into the assembly language

program. Executing the modified program with sample input data produced the instruction trace.

The traces consist of the IMPACT assembly instructions (LCODE [3]) which is similar to the MIPS

R2000 assembly language [32].

Since the performance number for many cache dimensions are needed, a one pass cache simulator

is used. The cache simulator for the experiments uses the recurrence/conflict model [17], where

only one pass over the instruction trace is needed to simulate all cache dimensions. Similarly,

the information required to derive miss penalty with load forwarding is collected for all cache

dimensions. In this paper, associativity of one-way, two-way, four-way, and fully-associative are

simulated. The block sizes considered are 16, 32, 64, and 128 bytes. The cache sizes range from

1K to 128K bytes.

4.3 Empirical Data and Analysis

For the purpose of experimentation, the code expanding optimizations described in Section 3 are

organized into four optimization levels with increasing functionality: *no* (no code expanding op-

timization), *pl* (instruction placement), *in* (function inline expansion plus instruction placement),

---

LCODE documentation is available as an internal report.

program	no	pl	in	su
cccp	-	2%	36%	54%
eqntott	-	1%	2%	7%
espresso	-	1%	10%	60%
mpla	-	1%	13%	41%
tbl	-	3%	22%	67%
xlisp	-	1%	18%	49%
yacc	-	4%	21%	110%
average	-	2%	17%	55%

Table 3: Accumulated code size increase.

and *su* (superscalar optimization, function inline expansion, and instruction placement). Experiments are conducted by varying the optimization level to measure the incremental and accumulative effects of these optimizations.

General Effects

In order to quantify the effect of optimization on code size, the object code size was measured for each level of optimization. Table 3 shows the relative object code size for each optimization level. All ratios and percentages are computed based on the code size without code expanding optimization. Instruction placement increases the average code size by 2%. Function inline expansion results in a 15% code expansion after instruction placement, as indicated by the 17% increase in average code size in the *in* column of Table 3. Superscalar optimization further increases the code size by 38% after both inline expansion and instruction placement. The total code expansion due to all the three optimizations is 55%, which reinforces the concern that these optimizations may degrade the instruction cache performance.

The instruction working set of a program is defined as the smallest fully-associative instruction cache which achieves a 0.1% miss ratio for the program. It provides a relative measure of cache

program	16 byte block				32 byte block				64 byte block				128 byte block			
	no	pl	in	su	no	pl	in	su	no	pl	in	su	no	pl	in	su
cccp	13	13	13	13	13	13	12	13	12	12	12	13	13	12	12	13
eqntott	10	10	10	10	10	10	9	10	10	10	10	10	11	11	11	11
espresso	14	14	14	15	14	14	14	15	13	13	13	14	13	13	13	14
mpla	14	13	14	15	14	13	14	15	14	14	14	15	14	14	14	15
tbl	14	14	15	15	14	14	15	15	14	14	15	15	14	13	14	15
xlisp	12	12	13	13	13	12	13	13	13	13	13	14	13	13	13	14
yacc	11	11	12	13	12	11	11	13	11	11	11	13	11	11	11	13

Table 4: Working set size for various block sizes in log_2 cache size.

program	no		pl		in		su	
	num	% inc	num	% inc	num	% inc	num	% inc
cccp	5.1	-	7.5	47	7.7	50	10.5	105
eqntott	3.8	-	5.9	53	5.9	54	5.9	54
espresso	6.4	-	8.4	31	9.1	42	14.8	131
mpla	5.1	-	8.9	76	9.9	96	17.8	253
tbl	3.5	-	4.9	42	6.4	84	13.1	278
xlisp	4.2	-	6.3	50	9.5	129	10.8	159
yacc	4.0	-	5.9	47	6.1	51	13.0	223
average	4.6	-	6.8	48	7.8	70	12.3	167

Table 5: Average number of sequential instructions.

size requirement by programs. Table 4 presents the instruction working set size of each benchmark for all optimization levels. All numbers presented are in log_2 scale (e.g.. 14 is a 16K byte cache). The largest working set size needs at most a 32K byte cache. All miss ratios for the larger caches are considered negligible, and for this reason. cache sizes larger than 32K will generally not be shown in this paper. Instruction placement and function inline expansion have very little effect on the instruction working set size. Superscalar optimization approximately double the instruction working set size. This is expected since superscalar optimizations results in the largest increase in code size.

program	base no	% change		
		pl	in	su
cccp	2.89×10^7	-0.27	-2.01	-3.17
eqntott	1.47×10^8	-0.42	-0.43	-0.45
espresso	5.48×10^7	+0.18	-1.23	-3.33
mpla	1.07×10^8	-0.62	-6.18	-10.1
tbl	3.08×10^7	+0.21	-12.3	-16.2
xlisp	1.46×10^8	-1.84	-14.6	-16.7
yacc	3.47×10^7	-1.00	+0.13	+6.53

Table 6: Number of dynamic references.

As discussed in Section 3, all of the three code expanding optimizations can improve the sequentiality of instruction access. To quantify this effect, the average number of sequential instructions executed between taken branches was measured. As shown in Table 5, all of the three optimizations improve the sequentiality significantly. With all optimizations, the average number of sequential instructions increased from 4.6 to 12.3. This dramatic increase in sequentiality suggests that schemes such as load forwarding may be able to offset the negative effect of code expansion. We will further explore this subject later in this section.

Although the static code size increases significantly after the code expanding optimizations, the number of dynamic instruction references tends to decrease with each additional level of optimizations. Table 6 presents the number of instruction references for each benchmark program. The largest improvement results from function inline expansion; this is due to the increasing opportunity to apply classic local and global optimizations on the inlined version of the code and to eliminate instructions that save and restore registers across function boundaries. The purpose for superscalar optimizations is to uncover parallelism and scheduling opportunities. Note however, that superscalar optimizations often result in a decrease in the number of instruction references. The contribution of instruction placement to the number of dynamic references is small when compared

20

program	16 byte block				32 byte block			
	no	pl	in	su	no	pl	in	su
cccp	840	800	890	1120	450	430	480	590
eqntott	400	500	400	500	200	300	200	200
espresso	2170	2170	2320	3290	1140	1130	1210	1740
mpla	3500	3300	4200	5620	1900	1700	2200	2970
tbl	1310	1270	1510	2000	690	660	780	1070
xlisp	800	700	800	1100	400	400	500	600
yacc	980	910	1040	2020	530	480	550	1060
	64 byte block				128 byte block			
cccp	240	230	260	310	140	130	140	170
eqntott	100	200	100	100	90	100	100	90
espresso	600	600	640	940	320	330	350	520
mpla	1000	900	1200	1600	600	500	700	870
tbl	360	350	420	570	180	180	220	300
xlisp	300	300	300	300	200	200	200	200
yacc	290	250	300	570	160	130	160	310

Table 7: Number of unique references.

to the other optimizations since instruction placement only performs code reordering.

The sum of the number of *recurrent* references and the number of *unique* references constitutes the number of total dynamic references. Table 7 shows that the number of *unique* references increases for inlining and superscalar optimizations. but decreases for instruction placement. The absolute difference within the *unique* references does not constitute a significant variation in the miss ratio since the difference is insignificant when compared to the number of dynamic references in Table 6.

Instruction Placement
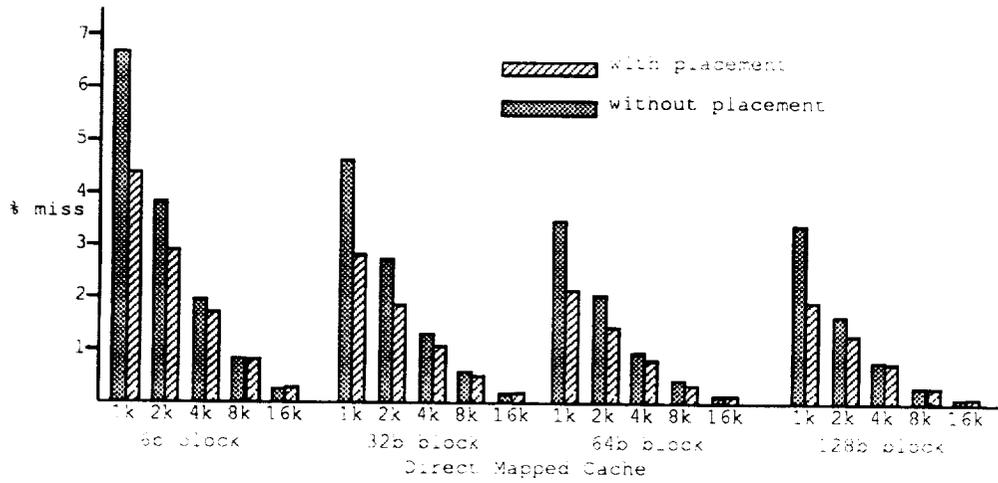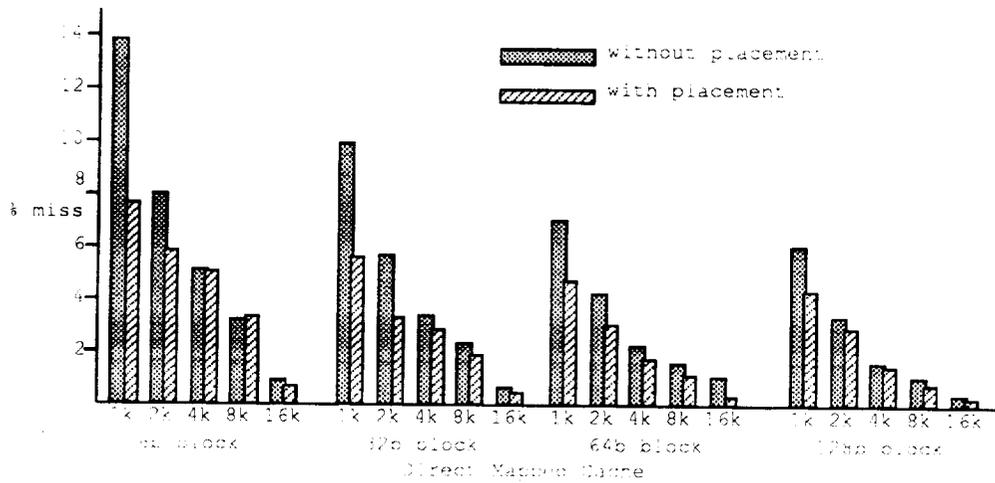
Figure 5: Average effect of placement.

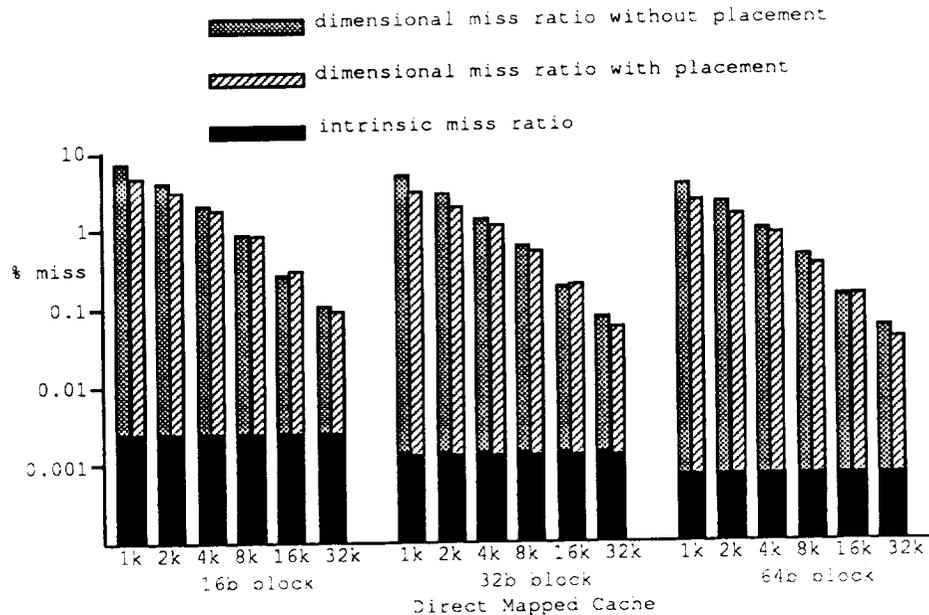Figure 6: The effect of placement for the highest miss ratios.

Figure 7: Effect of placement on dimensional conflicts and unique references.

Figure 5 shows the effect of instruction placement on the average cache miss ratio [4]. On one hand, instruction placement reduces miss ratio for small caches (1K and 2K). For example, the miss ratio of a 1K cache with placement is comparable to that of a 2K cache without placement. On the other hand, instruction placement has very little effect on large caches (8K and 16K). The same trend can be observed from the worst case miss ratios in Figure 6. The worst case miss ratio is the maximal miss ratio observed among all benchmark programs. Note that the benefit of instruction placement is more pronounced for programs with high miss ratios. This is a very desirable effect since it increases the stability of the cache performance.

To analyze why instruction placement improves the performance of small caches, we have measured the misses due to unique references (intrinsic misses, see Section 2) and those due to dimensional conflicts (dimensional misses). The log plot of Figure 7 shows the contribution of each to

---

[4] We found that the effect of instruction placement on the cache miss ratio of other associativities closely follows the trend of the direct mapped cache case, therefore only the direct mapped cache results are presented.

the miss ratio with and without placement. The black bars show the intrinsic miss ratio. Figure 7 clearly indicates that instruction placement makes negligible difference in the number of intrinsic misses [5]. The shaded bars in Figure 7 show the dimensional misses. As can be seen in the figure, the reduced miss ratio after placement is due to decreased dimensional conflicts [6].

The changes in program behavior due to instruction placement explain the discrepancy between small and large caches. The working set of the benchmark programs do not fit into small caches. This accounts for the high miss ratio of the small caches. Instruction placement separates the frequently executed code segments from those executed infrequently. This helps the small caches to accommodate the frequently executed portions of the programs. Therefore, the performance of small caches improves significantly after instruction placement. Since large caches can accommodate the working set of most benchmark programs, the compaction effect of instruction placement does not make a significant difference for these cache sizes.

Function Inline Expansion

Function inline expansion has two conflicting effects on cache performance. On the positive side, with inlining the caller and callee bodies are processed together by instruction placement. This allows instruction placement to significantly increase the sequentiality of the program (see Table 5). When the cache miss ratio is high, the increased sequentiality reduces the miss ratio because it increases the number of useful bytes transferred for each cache miss. On the negative side, inlining increases the working set size (see Tables 3 and 4). If the working set fits into a cache before inlining

---

[5] The reader is encouraged to derive the intrinsic miss ratio by dividing the number of unique references in Table 7 with the number of dynamic references in Table 6.

[6] Note that Figure 7 is in log scale, which is necessary to make the intrinsic miss ratio visible. However, the log scale also magnifies the miss ratio of large caches. For example, instruction placement seem to make comparable difference for small caches (1K and 2K) and large caches (16K and 32K) in Figure 7. However, it is clear from Figure 7 that instruction placement has strong effect on small caches but negligible effect on large caches.
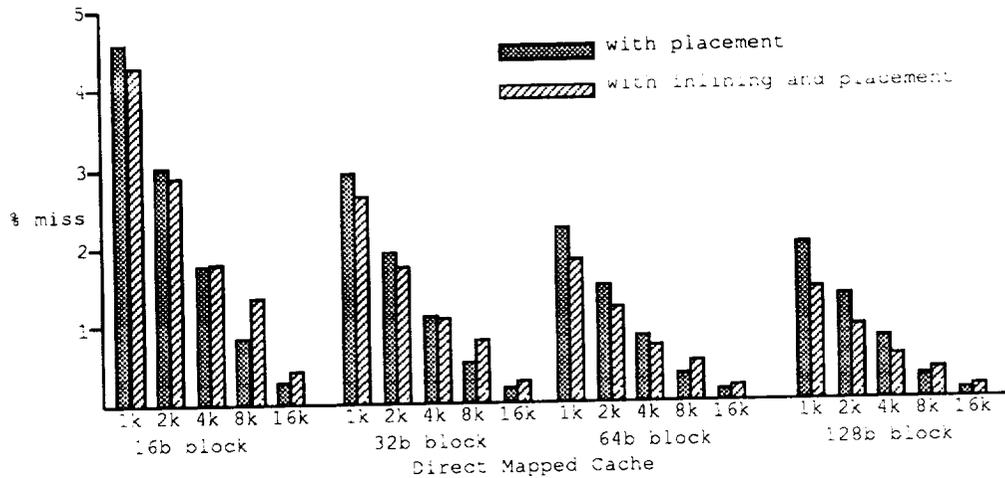
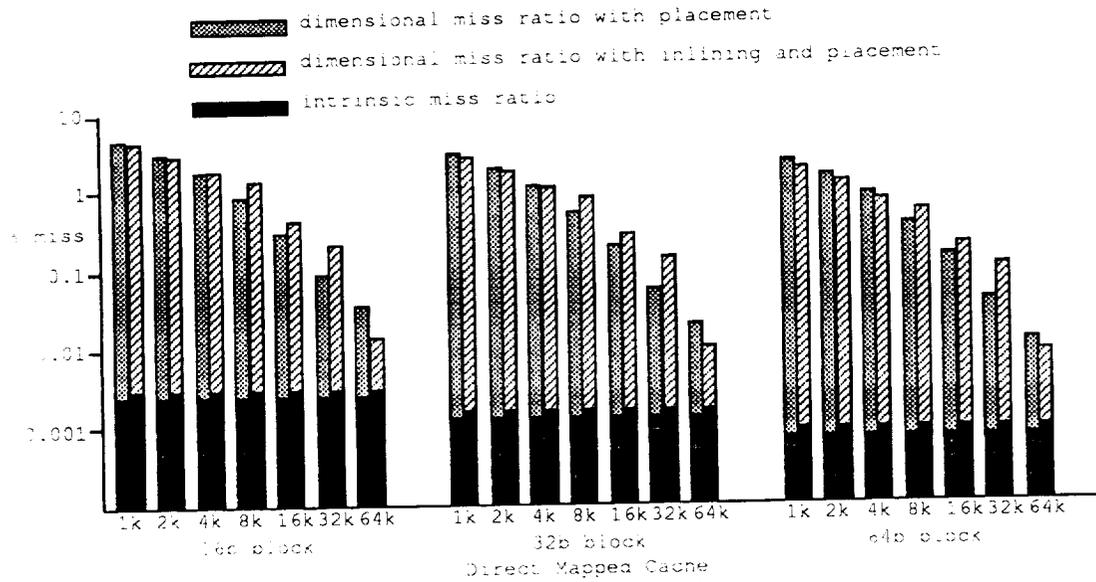Figure 8: Average effect of inlining and placement.

Figure 9: Effect of inlining and placement on dimensional conflicts and unique references.
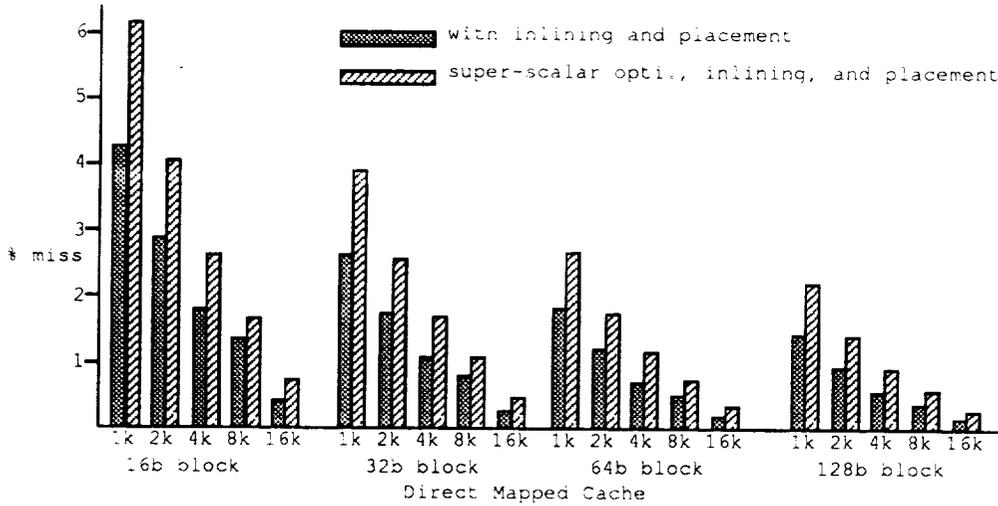
Figure 10: Effect of superscalar optimizations for direct mapped cache.

but does not after inlining, the cache miss ratio may increase substantially.

Figures 8 and 9 show the effect of inline function expansion on cache performance [7]. The cache miss ratio is relatively high for small caches before inlining. In this range, the increased sequentiality reduces the cache miss ratio. In the middle range (8K, 16K, and 32K), the working sets of some benchmarks fit in the cache before inlining but not after inlining. As a result, inlining increases cache miss ratio. The 64K cache is large enough to accommodate the program working set before and after inlining. Therefore, inlining has negligible effect in caches of size 64K and greater.

Superscalar Optimizations

Figure 10 shows the changes in the cache miss ratios when superscalar optimizations are applied after inlining and placement. The miss ratios are consistently higher with superscalar optimizations. Therefore, a larger cache is required to compensate for the effect of superscalar optimizations to maintain the same miss ratio. This information is consistent with the working set size calculated in

---

[7] As before, the trend for higher set associativities is very close to the results for direct mapped cache. Thus, only the direct mapped results are presented.
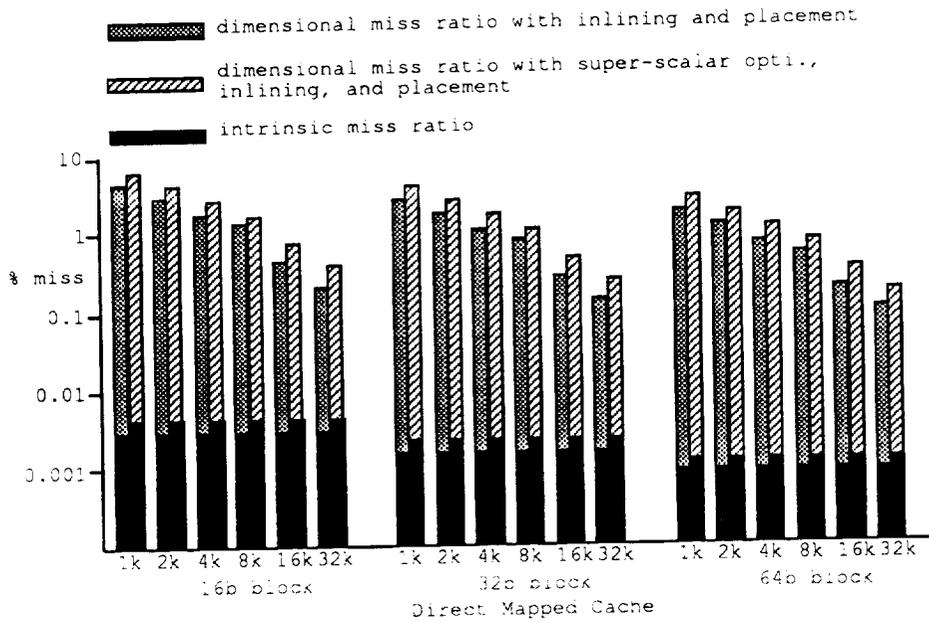
Figure 11: Effect of superscalar optimizations on dimensional conflicts and unique references.

Table 4. If the block sizes are kept constant, the required cache size to maintain the same level of miss ratio is approximately twice the cache size over that of code with no superscalar optimizations.

Figure 11 indicates that superscalar optimizations increase the number of unique references, but the increase is not significant. Therefore, it is the increase in code size rather than the increase in *unique* references that is the primary cause of reduced cache performance.

All Optimizations

Figure 12 shows the cumulative effect of all optimizations on direct mapped caches. Intuitively, smaller caches should perform worse on expanded code because of increase in the expected number of dimensional conflicts. However, the experimental data show the opposite. For the 1k and 2k caches, the miss ratio of code without code expanding optimizations are larger than the miss ratios of code with code expanding optimizations. Sequentiality is increased by superscalar optimizations, thus for larger block size, the decrease in miss ratio is due to sequentiality (e.g., for 1k cache in
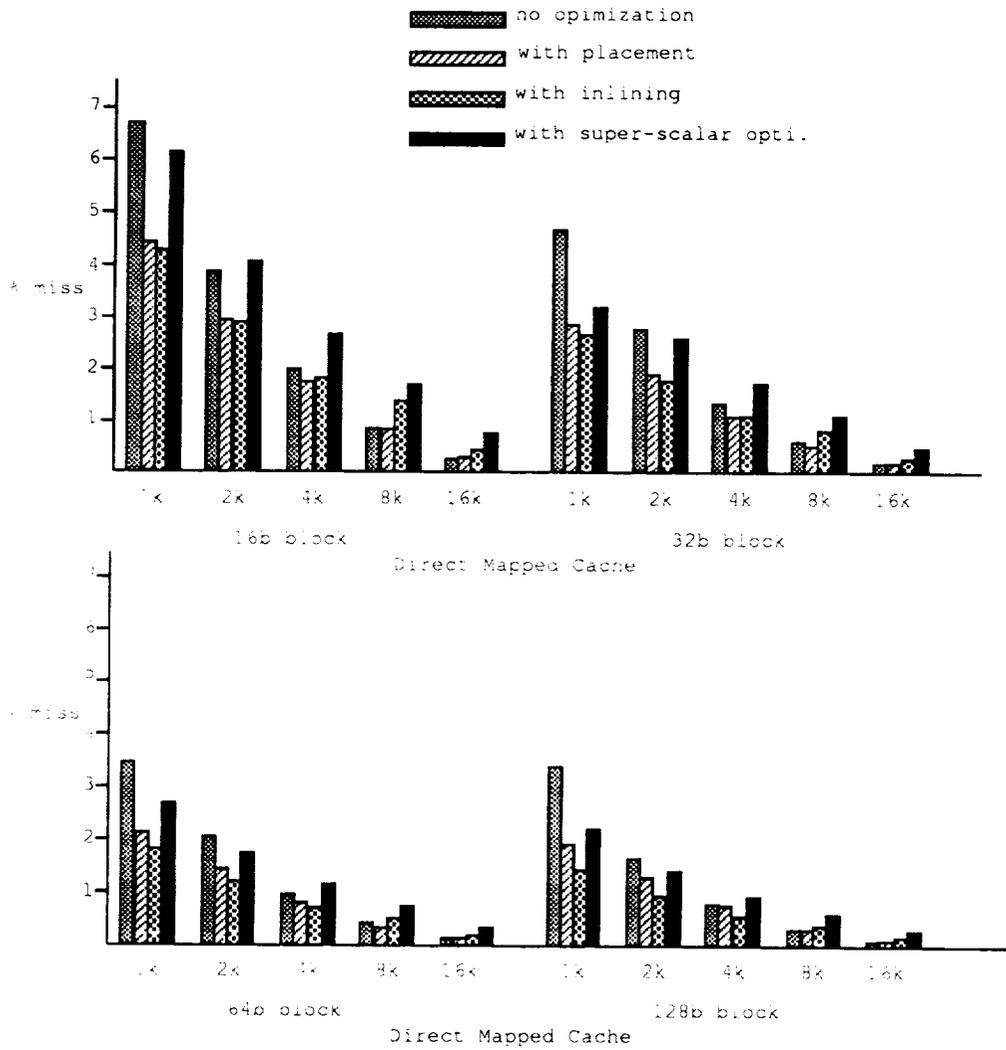
Figure 12: Cumulative effect of all optimizations for direct mapped cache.

Figure 12. code with superscalar optimizations has a larger drop in miss ratio going from 64B to 128B block size than code with no optimization). For small block sizes, the positive effect of higher sequentiality disapears, and the negative effect of code expansion causes an increase in the miss ratio. However, the increase in code locality by function inlining and instruction placement is still large enough to offset the negative effect of the code expansion, and a slight decrease in the miss ratio can still be seen in small caches.

Load Forwarding

The results of load forwarding are presented in Figure 13. Since superscalar optimizations have the worst results thus far, they are used here to evaluate the effectiveness of load forwarding. The initial memory repair latency (L) is assumed to be 4 cycles, and the cache-memory bandwidth (3) is assumed to be 4 bytes. Equations 3 and 4 are used to calculate the relative miss time penalty. Load forwarding reduces the miss penalty and effectively upgrades the cache to a performance level similar to a non load-forwarding cache of twice the size. For example, assume that 2K direct mapped cache with block size of 64 bytes is used with load forwarding. Using the same block size, the miss penalty is approximately the same as that of a 4K cache without load forwarding. When superscalar optimizations are used, the designer can either double the cache size to maintain the same performance level or use load forwarding and achieve the same result.

Another observation is that a block size of 128 bytes has consistently higher average miss penalties than for other block sizes. This can be explained by the number of sequential instructions shown in Table 5. The overall average run length for superscalar optimizations is approximately 12.3 instructions (49.2 bytes). It is possible that the first non-sequential miss will not be in the beginning of the block (see Figure 14). By using the symbol R for the run length, and l as the run
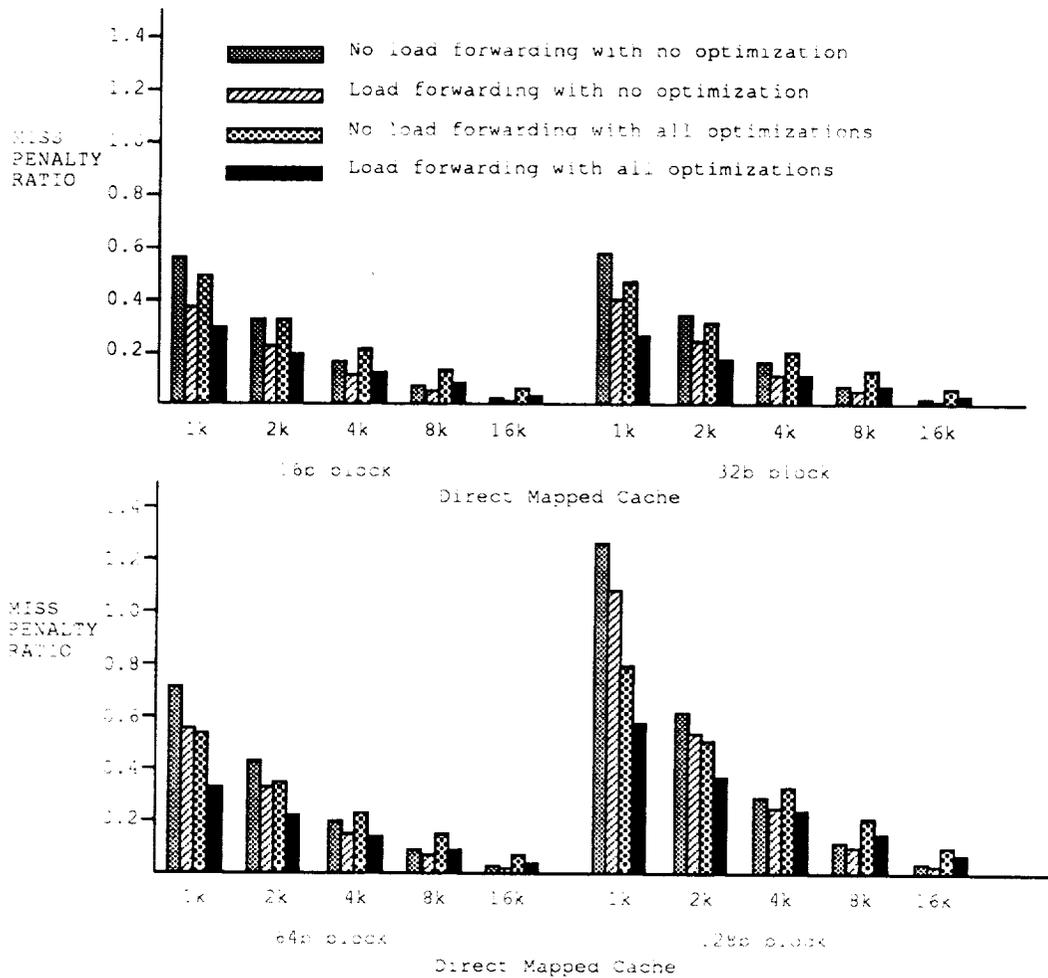
29

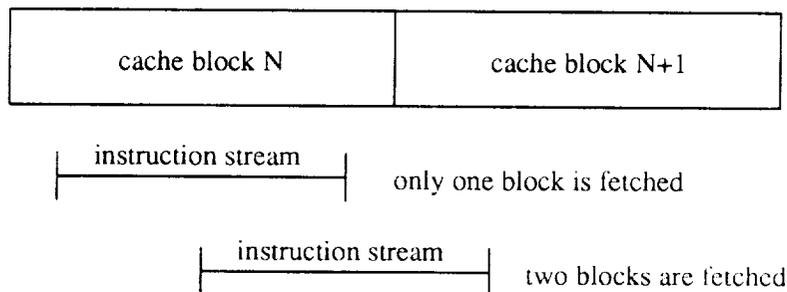Figure 13: Effect of load forwarding for direct mapped cache.

Figure 14: Reference stream and cache block refills.

length starting location within the cache block, the total number of cache blocks involved in a miss is formulated as,

$$\Gamma(l, B, R) = \lceil \frac{(l + R)}{2^B/\beta} \rceil.$$ (6)

The ceiling function is used to include all used cache blocks. For each run length, there are $2^B/\beta$ starting locations. Assuming uniform distribution for all starting locations, the probability of each starting location would be $\beta/2^B$. Therefore, the penalty of each cache miss for a particular run length is shown as Equation 7.

$$P(R, B) = \sum_{l=0}^{\frac{2^B}{\beta}-1} \frac{1}{2^B/\beta} \times \{\Gamma(l, B, R) \times (L + \frac{2^B}{\beta}) - R\}$$ (7)

For simplicity, an integer approximation of the run length is used. Instead of 12.3, the value of 13 is used for R in Equations 6 and 7.

$$P(13, 4) = 19 \ cycles$$ (8)

$$P(13, 5) = 17 \ cycles$$ (9)

$$P(13, 6) = 22 \ cycles$$ (10)

$$P(13, 7) = 36.5 \ cycles$$ (11)

The calculated values follow the trend in Figure 13 closely. For B equal to 4, 5, and 6, the load forwarding miss penalties are relatively the same, with B equal to 5 (the lowest), and B equal to 4 (the next lowest). For B equal to 7, the load forwarding miss penalty is noticeably higher than the other block sizes, and this can also be shown by using Equation 7.

The miss penalty for each run of sequential accesses is dominated by three values: the initial load delay, the number of refill cycles with load forwarding, and the number of refill cycles without load forwarding. While the initial load delay is dependent upon the hardware design technology, the
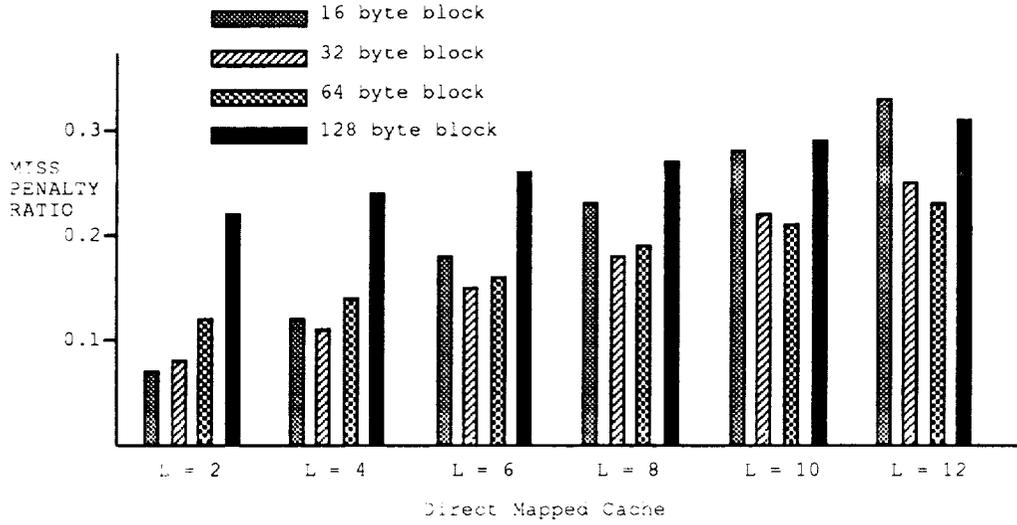
31

Figure 15: Effect of initial load delay (4k cache).

non-stalling and stalling refill cycles are related to the block size and the instruction sequentiality.

Before the initial load delay reaches a certain threshold value, the number of refill cycles will have a

dominant effect upon the miss penalty. Larger block sizes will tend to have higher wasted number

of refill cycles than smaller block sizes. However, larger block sizes are penalized less for the initial

load delay than smaller block sizes. Figure 15 shows the effect of varying the value of the initial

load delay on block sizes for a 4k cache. For each value of L, the miss penalty ratio is compared

between four block sizes. For small values of L, 16 and 32-byte blocks perform the best. But for

larger values of L, 64-byte block performs the best. This is also verified by Equation 7. Here, the

value of L is set to 10.

$$P(13, 4) = 43 \ cycles \qquad (12)$$

$$P(13, 5) = 32 \ cycles \qquad (13)$$

$$P(13, 6) = 32.5 \ cycles \qquad (14)$$

$$P(13, 7) = 44.75 \ cycles \qquad (15)$$

32

From Figure 15, for initial delay of 10, block sizes of 32 and 64 bytes have similar performances, and block sizes of 16 and 128 bytes have similar performances.

As the value of L increases, the performance of the larger block sizes increases while the performance of the smaller block sizes decreases. It is not until an initial load delay of 40 cycles before 128-byte blocks start to out-perform other block sizes. For smaller cache sizes, the miss ratios are the dominating factor, and a smaller block size should be used. On the contrary, for larger cache sizes, since the miss ratios are very small, larger block sizes are preferred.

5 Conclusions

This paper analyzes the effect of compile-time code expanding optimizations on instruction cache design. We first show that instruction placement, function inline expansion, and superscalar optimizations cause substantial code expansion, reinforcing the concern that they may increase the cache size required to achieve a given performance level. We then show the actual effect of each optimization on cache design.

Among the three types of optimizations, instruction placement causes the least amount of code expansion. Its effects on the cache performance are mostly due to the increased instruction access sequentiality. For small caches where the miss ratio is relatively high, the increased sequentiality reduces the number of cache misses by increasing the useful bytes transferred for each cache miss. For large caches where the miss ratio is relatively low, the effect of instruction placement is negligible.

Inline function expansion affects the cache performance by increasing both the sequentiality and the working set size. For small caches where the miss ratio is high, the increased sequentiality

33

helps to reduce the miss ratio. Due to the increased working set size, some benchmarks which fit into moderately sized caches before inlining do not fit after inlining. Therefore, inlining increases the miss ratio of moderately-sized caches. For large caches, since the working sets fit in the cache before and after the cache, the effect of inlining is insignificant.

Superscalar optimizations increase the cache size required for a given miss ratio. However, they increase the sequentiality of instruction access so much that a simple load-forward scheme effectively cancels the negative effects. Using load forwarding, the three types of code-expanding optimizations jointly improves the performance of small caches in spite of the substantial code expansion. Load forwarding also allows the code expanding optimization to have little negative effect on the performance of large caches.

Acknowledgements

References

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

[2] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc., 1989.

[3] MIPS Computer Systems, *MIPS language programmer's guide*, 1986.

[4] P. P. Chang and W. W. Hwu, "Inline function expansion for compiling C programs," in *Proc. 1989 ACM Conf. on Prog. Lang. Design and Implementation*, (Portland, OR), June 1989.

[5] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[6] A. J. Smith, "Line (block) size choice for CPU cache memories," *IEEE Trans. Computers*, vol. C-36, pp. 1063–1075, Sept. 1987.

[7] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Computers*, vol. C-38, pp. 1612–1630, Dec. 1989.

[8] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Trans. Computers*, vol. C-34, pp. 234–241, Mar. 1985.

[9] R. J. Eickenmeyer and J. H. Patel, "Performance evaluation of on-chip register and cache organizations," in *Proc. 15th Ann. Int'l Symp. Computer Architecture*, (Honolulu, Hawaii), pp. 64–72, May 1988.

[10] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," in *Proc. 15th Ann. Int'l Symp. Computer Architecture*, (Honolulu, Hawaii), pp. 290–298, June 1988.

[11] D. B. Alpert and M. J. Flynn, "Performance trade-offs for microprocessor cache memories," *Micro*, pp. 44–54, Aug. 1988.

[12] M. D. Hill and A. J. Smith, "Experimental evaluation of on-chip microprocessor cache memories," in *Proc. 11th Ann. Int'l Symp. Computer Architecture*, (Ann Arbor, MI), pp. 158–166, June 1984.

[13] J. Davidson and R. Vaughan, "The effect of instruction set complexity on program size and memory performance," in *Proc. Second Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Palo Alto, CA), pp. 60–64, Oct. 1987.

[14] C. L. Mitchell and M. J. Flynn, "The effects of processor architecture on instruction memory traffic," *ACM Transaction on Computer Systems*, vol. 8, pp. 230–250, Aug. 90.

[15] P. Steenkiste, "The impact of code density on instruction cache performance," in *Proc. 16th Ann. Int'l Symp. Computer Architecture*, (Jerusalem, Israel), pp. 252–259, June 1989.

[16] K. J. Cuderman and M. J. Flynn, "The relative effects of optimization on instruction architecture performance," Tech. Rep. CSL-TR-89-398, Computer Systems Laboratory, Stanford University, Stanford, CA, Oct. 1989.

[17] T. M. Conte and W. W. Hwu, "Single-pass memory system evaluation for multiprogramming workloads," Tech. Rep. CSG-122, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1990.

[18] W. W. Hwu and T. M. Conte, "The susceptibility of programs to context switching," *IEEE Trans. Computers*, 1991, submitted for publication.

[19] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. Int'l Symp. Computer Architecture*, (Jerusalem, Israel), pp. 242–251, June 1989.

[20] S. McFarling, "Program optimization for instruction caches," in *Proc. Third Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 183–191, Apr. 1989.

[21] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. 1990 ACM Conf. on Prog. Lang. Design and Implementation*, (White Plains, NY), June 1990.

[22] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Computers*, vol. c-30, no. 7, pp. 478–490, July 1981.

[23] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Ann. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[24] R. Allen and S. Johnson, "Compiling C for vectorization, parallelism, and inline expansion," in *Proc. 1988 ACM Conf. on Prog. Lang. Design and Implementation.* (Atlanta, Georgia), June 1988.

[25] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," in *Proc. ACM SIGPLAN '82 Symp. Compiler Construction*, 1982.

[26] F. Chow and J. Hennessy, "Register allocation by priority-bases coloring," in *Proc. ACM SIGPLAN '84 Symp. Compiler Construction*, pp. 222–232, 1984.

[27] J. R. Ellis, *Bulldog: a Compiler for VLIW Architectures*. Combridge, MA: The MIT Press, 1986.

[28] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecuture for high performance scientific computing," in *Proc. 14st Ann. Workshop on Microprogramming and Microarchitectures*, Oct. 1981.

[29] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," in *Proc. Second Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, Oct. 1987.

[30] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "Impact: An architectural framework for multiple-instruction processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), June 1991.

[31] "Spec newsletter," Feb. 1989. SPEC, Fremont, CA.

[32] G. Kane, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1988.