DATA PRELOAD FOR SUPERSCALAR AND VLIW PROCESSORS

BY

WILLIAM YU-WEI CHEN, JR.

B.S., The Ohio State University, 1988
M.S., University of Illinois at Urbana-Champaign, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

DATA PRELOAD FOR SUPERSCALAR AND VLIW PROCESSORS

William Yu-Wei Chen, Jr., Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1993
Wen-mei W. Hwu, Advisor

Processor design techniques, such as pipelining, superscalar, and VLIW, have dramatically decreased the average number of clock cycles per instruction. As a result, each execution cycle has become more significant to overall system performance. To maximize the effectiveness of each cycle, one must expose instruction-level parallelism and employ memory latency tolerant techniques. However, without special architecture support, a superscalar compiler cannot effectively accomplish these two tasks in the presence of control and memory access dependences.

Preloading is a class of architectural support which allows memory reads to be performed early in spite of potential violation of control and memory access dependences. With preload support, a superscalar compiler can perform more aggressive code reordering to provide increased tolerance of cache and memory access latencies and increasing instruction-level parallelism. This thesis discusses the architectural features and compiler support required to effectively utilize preload instructions to increase the overall system performance.

The first hardware support is *preload register update*, a data preload support for load scheduling to reduce first-level cache hit latency. Preload register update keeps the load destination registers coherent when load instructions are moved past store instructions that reference the same location. With this addition, superscalar processors can more effectively tolerate longer data access latencies.

The second hardware support is *memory conflict buffer*. Memory conflict buffer extends preload register update support by allowing uses of the load to move above ambiguous stores. Correct program execution is maintained using the memory conflict buffer and repair code

provided by the compiler. With this addition, substantial speedup over an aggressive code scheduling model is achieved for a set of control intensive nonnumerical programs.

The last hardware support is *preload buffer*. Large data sets and slow memory sub-systems result in unacceptable performance for numerical programs. Preload buffer allows performing loads early while eliminating problems with cache pollution and extended register live ranges. Adding the prestore buffer allows loads to be scheduled in the presence of ambiguous stores. Preload buffer support in addition to cache prefetching support is shown to achieve better performance than cache prefetching alone for a set of benchmarks. In all cases, preloading decreases the bus traffic and reduces the miss rate when compared with no prefetching or cache prefetching.

ACKNOWLEDGMENTS

Discussions with Professor Wen-mei Hwu have always given me insight into the problems I am attempting to solve. He not only guided me through my research difficulties, but his personal advice has also nurtured and helped me to grow as an individual. At various times, Sabrina Hwu's laughter and great cooking helped tremendously. I thank them both for putting up with me for five years.

Of my colleagues, Scott Mahlke, was most helpful in giving suggestions, most of which were useful. Visitor and scholar rolled into one, Takuzo Kiyohara's two short years of stay advanced my research greatly. Ex-colleague, now professor at the University of South Carolina, Tom Conte, was alway willing to help out. I am grateful for his effort at setting up interviews for me when I had no job. Roger Bringmann, Dave Gallagher, John Gyllenhaal, and Rich Hank were very helpful in setting up the simulation environment for me. Sadun Anik, Grant Haab, and Nancy Warter were invaluable throughout the years. I extend gratitude to Jaushin Lee and his wife Belinda for helping me during times of personal need.

I wish to thank my friend Jackie Liu. For everything she did, I will always remember.

Lastly, but certainly not the least, I wish to extend my gratitude to my parents. They made me wake up early every Monday to call home. Certainly, that has always made Monday the longest and most fruitful working day.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

High-performance processors are often designed to exploit instruction level-parallelism (ILP). Superscalar and VLIW processors perform such a task by duplicating data paths and functional units, allowing them to issue multiple instructions per clock cycle. To assist the hardware to achieve its performance objectives, compilers for superscalar and VLIW processors use optimization and code scheduling techniques to expose simultaneously executable instructions to the hardware. To be effective, the scheduler must be able to freely reorder instructions within the scheduling scope. Compiler techniques, such as loop unrolling, register renaming, and critical path reduction, have been successful in removing register dependences within applications [1]. Aggressive branch handling techniques, such as branch target insertion, are utilized to allow the execution of multiple branches per cycle [2]. The combination of these optimizations gives the code scheduler more freedom to reorder instructions.

Unfortunately, the amount of static instruction reordering may be severely restricted due to dependences between memory instructions. Because memory references often occur on program critical paths, such loss of code reordering opportunities can limit the effectiveness of compile-time code scheduling. Therefore, high-quality superscalar and VLIW compilers usually employ sophisticated dependence analyzers to facilitate code reordering.

There are cases in which it is difficult for the dependence analyzer to determine the true relationship between reference pairs. For example, programming languages which allow data types such as unions and pointers pose difficulties for dependence analysis [3], [4]. The difficulty arises when several variables, or aliases, can access the same memory location. The possibility of a memory store aliasing with a memory load prevents these two instructions from being reordered. This limit to code reordering can in turn create two related problems. First, as load latency increases, the lack of independent instructions creates bubbles in the pipeline and decreases the throughput. Second, the execution critical path lengthened by the store to load memory dependence limits the amount of parallelization the compiler can perform.

This thesis addresses the problem of instruction reordering in the presence of memory dependences. A class of architecture support which we call data preloading is introduced to allow memory loads to be scheduled early within the static instruction sequence in spite of potential violation of memory access dependences. The thesis focuses on aspects of increasing ILP and tolerating memory access latencies, although the data preload concept can be applied to other optimizations as well. Compiler techniques are presented which take advantage of data preload. Architecture trade-offs are performed to evaluate the effectiveness of data preload.

1.2 Thesis Highlights

The following four topics are highlighted in this thesis.

First, memory dependence is identified as a major bottleneck in instruction scheduling for high-issue-rate processors. Historically, research in scheduling for ILP has been focussed primarily on moving instructions beyond conditional branches. The problem of conditional branches becomes less severe as the processor is able to provide higher issue bandwidth and

predicated execution [5]. As a result, the problem of extracting ILP increasingly involves the task of overcoming memory dependences. Architecture extension with data preload allows static instruction scheduling in the presence of memory dependence. Performance evaluations show data preload hardware gives additional speedup for high-issue-rate processors.

Second, the definition of speculative execution has been extended for data preload. Originally, speculative execution has been used in the literature to refer to executing an instruction before knowing that its execution is required. Data preload allows the execution of a memory load before a dependent memory store. A load and its dependent instructions thus executed may produce incorrect program states. The definition of speculative execution must include the execution of an instruction before knowing that it can be executed correctly. This thesis discusses and evaluates the implication of speculative instructions on execution correctness and performance.

Third, the design trade-offs for data preload are studied. The benefits of data preload are classified into three types: 1) reducing cache hit latency, 2) increasing ILP, and 3) reducing long memory access latency. The three proposed data preload hardware approaches for accomplishing these are 1) preload register update, 2) memory conflict buffer, and 3) preload buffer. Each hardware mechanism can have several possible designs. Each design can affect the implementation complexity and the range of benefits. The implication of several designs is presented.

Finally, the thesis looks at the problem of cache pollution when data are loaded into the cache too early. Early cache load is a product of cache prefetch, which is used to tolerate long memory access latency. During the course of program execution, the data location within the closest level of the memory hierarchy is difficult to determine statically. Cache pollution

results from the difficulty of determining prefetch distance. A complementary solution to cache prefetch, the preload buffer, is discussed and evaluated.

1.3 Organization of Thesis

This thesis is composed of 7 chapters. Chapter 2 overviews instruction scheduling and data preload instruction scheduling. An overview of previous work is also given. Chapter 3 presents a discussion of the IMPACT compiler and the simulation environments.

Chapter 4 discusses *preload register update*, a data preload support for load scheduling to reduce first-level cache hit latency. Preload register update allows the compiler to reorder load and store instructions even in the presence of inconclusive memory dependence analysis results. The new hardware support is described in terms of a detailed state diagram and comprehensive examples. An instruction scheduling algorithm is given to take full advantage of the data preload support.

Chapter 5 describes a hardware technique referred to as the *memory conflict buffer*. The main objective of the memory conflict buffer is to increase the opportunity of code reordering. This is achieved by removing the dependence between a store/load pair, allowing a memory load and its dependent instructions to be moved above any number of memory stores. Several alternative designs of the memory conflict buffer are discussed and evaluated.

Chapter 6 focuses primarily on data preload for simple loops in numerical benchmarks. Large data sets and slow memory subsystems result in unacceptable performance for this class of benchmark. The preload buffer allows performing loads early while eliminating problems

with cache pollution and extended register live ranges. Similarly, prestore buffer allows loads to be scheduled in the presence of ambiguous stores.

Chapter 7 contains concluding remarks and directions for future research.

CHAPTER 2

MEMORY DEPENDENCE AND INSTRUCTION SCHEDULING

2.1 Current Scheduling Paradigm

To understand the importance of data preload for high-issue-rate processors, one must first understand the static scheduling paradigm of compilers for contemporary processors. Instruction scheduling for today's processors must enforce a strict set of dependence constraints, most importantly register dependence and memory dependence. During scheduling, instructions are placed in an order determined by the processor resources available and, of course, dependence constraints. An instruction cannot be scheduled before the instruction it is dependent upon according to the rule of dependence. Therefore, instruction scheduling freedom is restricted by the number of dependences between instructions.

As the instruction issue rate of a processor increases, the number of independent instructions must also increase. Often the scheduler must look down past several conditional branches to find an independent instruction. Since memory load instructions often reside on program critical paths, it is imperative to schedule independent memory loads early so that other dependent instructions can be properly placed. However, a memory load cannot move above a memory store upon which it is dependent. To take advantage of increased instruction issue rate, the scheduler attempts to look further down the instruction stream for an independent instruction.

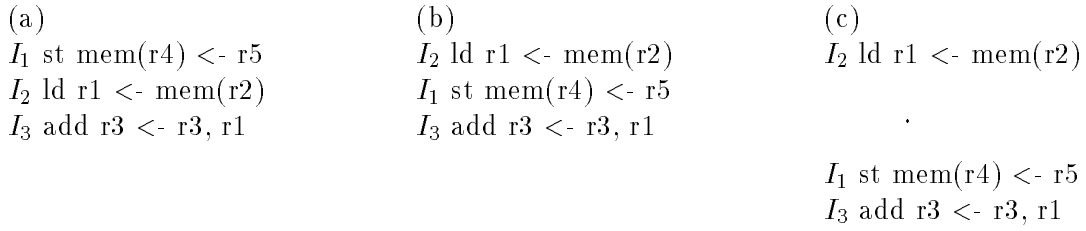


Figure 2.1 Illustrative example for instruction scheduling. (a) Unscheduled code, (b) scheduling for load delay, (c) scheduling for long load latency.

As the chance of memory dependence restricting code movement increases, the chance of finding independent instructions decreases.

2.1.1 The difficulties encountered by instruction scheduling

Figure 2.1(a) illustrates the basic problems involved in instruction scheduling. Assume that the load has a two-cycle latency for accessing the first-level cache, and the add has a one-cycle latency. If a dependence exists between I_1 and I_2, the shortest latency possible for these three instructions is four cycles. However if I_1 and I_2 are independent, it is possible to move I_2 before I_1, producing the schedule shown in Figure 2.1(b). The new schedule has a three-cycle latency because the load delay slot has been filled with an useful instruction. By moving I_2 before I_1, the flow dependent instruction I_3 may now move along with instruction I_2 also, thus creating more instruction scheduling freedom.

Instruction I_2 may not find its data in the first-level cache all the time. If I_2 misses in the cache fairly often, it is desirable to schedule the load early enough (Figure 2.1(c)) so that I_3 do not have to wait a long time before it can begin executing. The downside of scheduling a load too early is that register pressure has to be monitored carefully.

2.1.2 Memory dependence removal

Memory dependence may not alway exist between a pair of memory references. To be conservative initially, the compiler places dependences between a memory store and all other memory references. Then, the compiler must make careful judgement on whether a memory dependence can be removed. Such a decision is made by the memory dependence analyzer.

A memory dependence analyzer determines the relation between memory references. For array references, many algorithms exist to perform data dependence analysis [6], [7], [8]. Three possible conclusions can be reached regarding the relation between a pair of memory references: 1) they always access the same location; 2) they never access the same location; or 3) they may access the same location. In the first case, in which the two references alway access the same location, the compiler can choose to perform optimization to eliminate one of the memory references, thus eliminating the memory dependence. In the second case, in which the two references never access the same location, the compiler can remove the memory dependence between the pair of references. As a result, the instruction pair can be reordered freely. The third case occurs when either the dependence analyzer finds the reference pair to sometimes access the same location or it simply fails to prove that the two references are independent. In this case, the reference pair will be referred to as an *ambiguous reference pair*. If a memory store precedes a memory load and may access the same location as the load, the store will be referred to as an *ambiguous store* of the load.

It is in the third case that the memory dependence analyzer cannot remove the dependence between the reference pair. It is also clear that the conservative decisions made by the memory dependence analyzer are necessary to preserve correct program semantics. However, it is exactly this type of dependence that will prevent instruction reordering. During actual execution,

the compiler-marked dependent reference pair may only infrequently access the same memory location. A natural question is whether the instruction scheduler can optimize for the frequent instances in which the two references do not access the same location. This is the motivation for data preloading.

2.2 Scheduling with Data Preloading

Clearly, scheduling under the constraint of memory dependence can result in a less desirable instruction schedule for high-issue-rate processors. A way is needed to ignore certain memory dependences during instruction scheduling if it is profitable to do so. This thesis presents data preload as an architectural extension to existing processor architectures. The intention of data preload is to allow the compiler to ignore memory dependence during instruction scheduling, thereby increasing ILP and reducing memory latency effects.

The limitations with memory dependence can be removed by a combination of architecture and compiler support which we call data preloading. Data preloading allows the freedom of upward movement of memory load instructions while ignoring memory dependences. The instruction-set architecture is modified to incorporate a *preload* instruction. The preload instruction is scheduled by the compiler, and it utilizes the hardware support to allow correct program execution.

When a load is scheduled above an ambiguous store, the load becomes a preload. The preload instruction is not constrained in scheduling scope by memory dependence conditions discussed before. The hazard condition is detected by hardware support and corrected either by a hardware mechanism or a recovery code provided by the compiler. The scheduling scope of the instructions that use the preloaded data is different according to the architecture and compiler

support. To schedule a preload slightly earlier than the data use would eliminate the first-level cache hit latency. Scheduling the preload much earlier than the data use allows time for the accessed datum to traverse the memory hierarchy and bring the data to the CPU. When the instructions that use the data thus preloaded are scheduled with the preload above ambiguous stores, more aggressive instruction reordering is performed to increase ILP. By eliminating the need to observe memory dependence by the preload instruction, the instruction scheduler can better utilize available processor resources and better schedule for memory latencies.

2.3 Previous Work

2.3.1 Hardware-assisted alias resolution

A combined hardware and compiler scheme to keep in register a value that can be accessed via multiple aliases has been proposed by Dietz and Chi [9]. The register file is partitioned into several alias sets such that possibly aliased and simultaneously live references can reside in registers of the same alias set. A change in the content of one register will reflect in another register within the same alias set when their addresses are the same. An extension to this work has also been discussed in [10]. The purpose of this support is to enable the register allocator to keep values in the registers instead of reloading the values after each ambiguous store. There is no graceful way of controlling the coherence for instruction scheduling, and the size of the alias sets provided in the hardware implementation limits the amount of code reordering. Furthermore, there is no provision to allow code reordering of the instructions which use the result of the load. This is because the execution effect of these flow-dependent instructions cannot be reversed if an ambiguous store writes to the memory location referenced by the load.

2.3.2 Dynamic memory disambiguation

Nicolau describes dynamic memory disambiguation as a software solution to the problem of parallelization in the presence of ambiguous stores [11]. By inserting explicit address comparison and conditional branch instructions, dynamic memory disambiguation allows general code movement across ambiguous memory stores. However, the number of address comparison and conditional branch instructions inserted can be very large as a result of aggressive code reordering for control-intensive nonnumeric programs.

2.3.3 Dynamic scheduling

Out-of-order execution machines attempt to alleviate the problem by performing load bypassing. During dynamic execution, a memory load can bypass a memory store if their respective addresses are different. It has been shown that load bypassing is a major reason why dynamic code scheduling outperforms static code scheduling [12]. Using a hardware monitor as proposed by Emma et al., loads can bypass stores even when the store addresses are unknown [13]. The core of the monitor is similar to our proposed hardware scheme. However, the performance of load bypassing is constrained by the dynamic lookahead window size. Also, in the dynamic load bypassing model, the hardware support and the compiler support are considered separate entities. Thus, the compiler cannot utilize the hardware support of load bypassing to increase the opportunity for optimization and scheduling.

2.3.4 Nonblocking cache

For each cache miss, the CPU stalls a number of cycles required to access the memory hierarchy for the data. A nonblocking cache attempts to overlap the cache miss handling

with the execution of subsequent instructions [14], [15]. However, as the speed gap between a faster processor and a slower memory becomes more diverse, it is increasingly difficult to find independent instructions to perform useful computation.

2.3.5 Data cache prefetching

Data prefetching is an effective means of reducing the penalty of long memory access time beyond the primary cache [16]-[26].

Data prefetching is typically performed for scientific applications, in which the performance of caches is often inadequate. The idea of cache prefetching is to have the data available in the cache when the actual memory access occurs. Several prefetch strategies have been presented in the past. Some of these approaches use software support to issue prefetches, while others are strictly hardware based.

Hardware-based prefetch methods have been proposed to issue prefetches dynamically [16], [25], [24]. It can be as simple as implicit prefetching through a long cache block or as complicated as utilizing a separate data path for looking ahead in the instruction stream for potential prefetches. Two advantages are that hardware-based methods do not add instruction overhead to issue prefetches and are compiler independent. In predicting constant stride accesses, hardware-based methods perform very well; however, prediction based methods do not work well with nonconstant stride accesses. For hardware-based methods which depend on dynamic branch prediction, the prefetch scope is limited to a small number of branches.

Conventional means of compiler-based data prefetching require the compiler to generate a nonbinding memory load, *prefetch*, to a cache block [19]. This special prefetch instruction

informs the memory subsystem that a piece of data may be used in the near future. According to this hint, the data are fetched into the cache if not already present.

By placing the prefetched data into the cache, an increase in the data cache requirement is expected. Now, the cache holds not only the current working set, but also the future working set simultaneously. The working set size that the cache is required to hold depends on the prefetch strategy and the memory latency. If the working set requirement is larger than what the cache size can handle, cache pollution will occur, thus degrading the cache performance. In addition, as cache pollution increases, off-chip memory traffic also increases. The overall system performance can be degraded by prefetching if it is not carefully managed.

2.3.6 Data prefetch buffer

The problem of pollution is an important one and must be controlled for data prefetching to be an effective means of increasing the overall system performance. The use of prefetch buffer has been proposed by previous researchers [16], [22].

In [22], a fully associative prefetch buffer is used which can be thought of as a second cache. The data cache holds the current working set, and the prefetch buffer holds the possible future working set. When the data in the prefetch buffer are referenced, the associated cache line is transferred from the buffer into the data cache. The implementation of the prefetch buffer is complicated by the coherence protocol between the data cache and the prefetch buffer. Extra communication channels must exist between the cache and the prefetch buffer for the cache to inform the prefetch buffer of any dirty data and to transfer data from the buffer to the cache. These extra channels complicate cache controller design, especially for superscalar processors. A prefetch buffer in which each entry holds a decoded instruction is proposed by Lee, Yew, and

Lawrie [16]. The prefetches are generated for the source operands of the decoded instructions. Searching of the prefetch buffer is still necessary when executing a store so as not to prefetch a piece of stale data. When the store writes to the same location as a prefetch, the prefetch obtains the data from the store through a data forwarding mechanism.

CHAPTER 3

INSTRUMENTATION AND SIMULATION METHODS

All results in this work are obtained from simulations. Simulation offers a way to obtain the desirable design points for the actual hardware building. It is possible for a simulator to emulate nonexisting hardware. Simulation can also evaluate and debug many hardware models at a quick pace.

Two simulation methods are used within this thesis. The main differences between the two methods are simulation speed and detail. For a quick estimate of the program execution time on a standard pipelined processor with interlocking, the quicker first simulation method is used. The first method uses profiled branch statistics to estimate execution time without considering caching effects and other system I/Os. For more detailed studies such as memory subsystems and architecture variations, a program trace must be consumed by a simulator to gather the results. This chapter discusses the trace collection process and the two simulation methods in detail.

3.1 Trace Collection

This thesis uses the IMPACT-I C retargetable, optimizing compiler to perform translation of benchmark source into instrumented executables [27]. The IMPACT compiler uses several levels of intermediate representation, HCODE, LCODE, and MCODE. Each level of intermediate

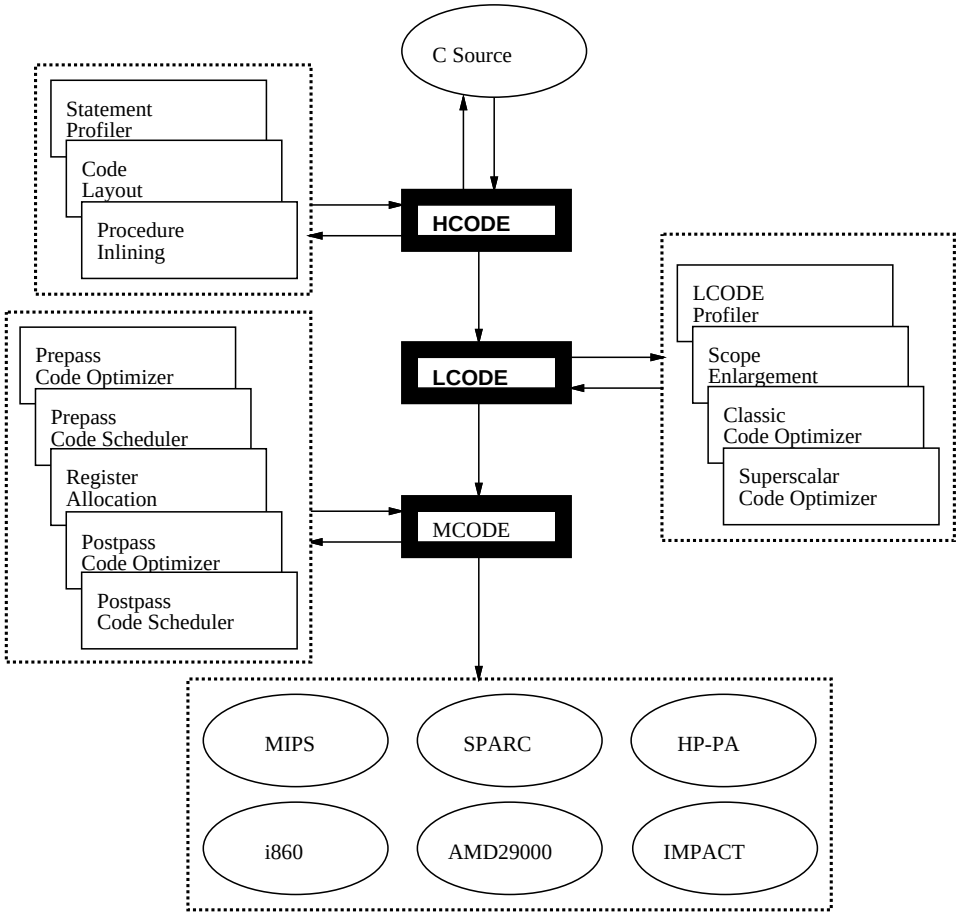


Figure 3.1 The IMPACT-I compiler overview.

representation is suited for a class of transformations. Figure 3.1 gives an overview of the IMPACT compiler and the modules associated with each intermediate representation.

The instrumentation code is inserted within the intermediate representation, MCODE, of the benchmark; MCODE is the closest intermediate instruction representation of the target architecture. Each MCODE instruction is later converted into an equivalent machine instruction for the target architecture. An example MCODE code for the C source in Figure 3.2(a) is shown in Figure 3.2(b).

Depending upon the run-time information needed, the compiler can insert the appropriate instrument code into MCODE to extract the information required to form a trace. Figure 3.3

(a)

```
int A[10];
main()
{
int i;
    for (i=0; i<10; i++)
      A[i] = 0;
    exit(0);
}
```

(b)

```
(function _main 1.000000 ())
  (cb 1 1.000000 ((flow 0 2 1.000000)))
      (Mop 1 prologue () ())
  (cb 2 1.000000 ((flow 0 3 1.000000)))
      (Mop 2 mov ((r 1 i)) ((l _A)))
      (Mop 3 add_u ((r 2 i)) ((i 40)(l _A)))
  (cb 3 10.000000 ((flow 1 3 9.000000)(flow 0 4 1.000000)))
      (Mop 4 st_i () ((i 0)(r 1 i)(mac gr0 i)))
      (Mop 5 add_u ((r 1 i)) ((i 4)(r 1 i)))
      (Mop 6 bne () ((r 1 i)(r 2 i)(cb 3)))
  (cb 4 1.000000 ((flow 1 5 1.000000)))
      (Mop 7 mov ((mac $P0 i)) ((mac gr0 i)))
      (Mop 8 jsr () ((l _exit)))
  (cb 5 1.000000 ())
      (Mop 9 epilogue () ())
      (Mop 10 rts () ())
(end _main)
```

Figure 3.2 (a) Original C source and (b) MCODE equivalent.

presents an example MCODE with the appropriate instrumentation code. Two Mop instructions, 4 (st_i) and 6 (bne), are extracted from Figure 3.2. Branching direction and virtual memory address are the two dynamic elements which are collected for the trace in this thesis. Branching direction trace is used to 1) generate the execution frequency of each basic block within each benchmark, and 2) combine with statically generated templates to generate a full instruction trace. A virtual memory address trace is used to provide the dynamic instruction sequence generated by the branch direction trace with the corresponding memory referencing pattern. The collected information is stored into a trace buffer. The trace buffer is flushed periodically to a simulator during benchmark execution.

3.1.1 Merging trace with MCODE

Within the branch direction trace, all conditional and unconditional branch instructions are monitored. By recording the branch frequency of these instructions, the execution frequency

(a)

```
(Mop 4 st_i () ((i 0)(r 1 i)(mac gr0 i)))                    # Store instruction
(Mop 11 mov ((mac $tr_temp i)) ((i 8)))                      # Store ID = 8
(Mop 12 st_i () ((i 0)(mac $tr_ptr i)(mac $tr_temp i)))      # Save store ID
(Mop 13 mov ((mac $tr_temp i)) ((i 0)))                      # Calculate
(Mop 14 add_u ((mac $tr_temp i)) ((mac $tr_temp i)(r 1 i))) # store address
(Mop 15 st_i () ((i 4)(mac $tr_ptr i)(mac $tr_temp i)))      # Save address
(Mop 16 add ((mac $tr_ptr i)) ((i 8)(mac $tr_ptr i)))        # Update trace buffer
```

(b)

```
(Mop 6 bne () ((r 1 i)(r 2 i)(cb 3)))                        # Branch instruction
(Mop 17 mov ((mac $tr_temp i)) ((i 5)))                      # Branch ID = 5
(Mop 18 st_i () ((i 0)(mac $tr_ptr i)(mac $tr_temp i)))      # Save branch ID
(Mop 19 add ((mac $tr_ptr i)) ((i 4)(mac $tr_ptr i)))        # Update trace buffer
```

Figure 3.3 Instrumented MCODE example: (a) Instrumentation for memory instruction and, (b) Instrumentation for branch instruction.

of all functions and basic blocks within each function can be calculated. For each branch instruction, both the execution frequency and the branch-taken frequency are recorded. This information is then used to reconstruct the profile data to be mapped back into the MCODE data structure.

The MCODE data structure has a frequency counter for each function and each branch instruction. After collecting the trace, the branch frequency can be mapped into the MCODE data structure. For example, the first floating-point number associated with function or cb in Figure 3.2(b) shows the execution frequency, and the flow associated with each cb shows the frequency of control transfer out. This information is used to perform quick estimates of the benchmark execution cycle; it will be discussed in detail in Section 3.2.

3.1.2 Merging trace with instruction templates

A dynamic instruction execution sequence can be constructed with the information gathered in the branch direction trace. This is performed in three steps. First, a static instruction

sequence for the benchmark is captured into an instruction template. For each instruction, the instruction template contains its opcode and its processor resources used. Second, the instrumented code is executed to gather the trace. Third, by using the branch direction information within the trace, the branch is indexed into the instruction template for a list of instructions within the basic block. Since entering a basic block implies that all the instructions within the basic block are to be executed, these instructions obtained from the instruction template represent the dynamically executed instructions. The instruction trace is now complete.

The virtual memory address trace is also merged into the instruction trace generated using the same steps. When encountering a memory instruction in the dynamic instruction sequence, the memory trace is consulted. Individual virtual addresses are taken from the memory trace in the order in which they are generated. Each memory instruction within the dynamic instruction trace now has its dynamic referencing virtual address.

3.2 Estimating Execution Cycles with Branch Profile

Based on the branch profile in the MCODE data structure, a quick estimate of the execution cycles for a given benchmark on an architecture can be calculated. This is accomplished in conjunction with an instruction scheduler. The instruction scheduler records static information regarding the reordered instruction sequence during scheduling, and the simulator calculates the benchmark execution time based on the recorded information.

This process is best illustrated with an example. Consider the control block (cb) shown in Figure 3.4. In this example, the control block consists of two basic blocks. In this representation, the control block can be entered only from the top but may leave at one or more exit points. The control entrance point begins with Mop 1, and the exit points consists of Mop 3 and 6. Static

```
(cb 2 1000.0
      ((flow 1 3 10.000000)
       (flow 1 2 970.000000)
       (flow 0 3 20.000000)))
  (Mop 1 add ((r 2 i)) ((r 2 i)(r 1 i)) (i_t 1))
  (Mop 2 add ((r 1 i)) ((r 1 i)(i -1)) (i_t 2))
  (Mop 3 beq () ((r 1 i)(i 0)(cb 3)) (i_t 3))
  (Mop 4 add ((r 2 i)) ((r 2 i)(r 1 i)) (i_t 4))
  (Mop 5 add ((r 1 i)) ((r 1 i)(i -1)) (i_t 5))
  (Mop 6 bne_fs () ((r 1 i)(i 0)(cb 2)) (i_t 6))
```

Figure 3.4 Example control block and scheduler directed issue time.

branch prediction dictates that Mop 3 is likely to fall through and Mop 6 is likely taken. The instruction scheduler inserts the *i_t* directive for each Mop instruction. The number associated with the directive indicates the time at which the instruction can be executed with respect to the control entrance point, in this case Mop 1. Assuming that the add and branch instructions take one cycle each to complete, it will take three cycles to complete Mops 1 to 3 and three cycles to complete Mops 4 to 6.

To compute the total time to execute the control block, the branch profile is used. Looking at the control block header, control block 2 is executed a total of a thousand times. Of the thousand times, the control block is exited at Mop 3 ten times, and falls through to the next control block at Mop 6 twenty times. Therefore, Mops 1 to 3 are executed a thousand times while Mops 4 to 6 are executed 990 times. This amounts to 5970 cycles to execute this control block using the branch profile. However, this execution time does not include the branch miss prediction penalty. Since Mops 3 and 6 are mispredicted ten and twenty times, respectively, a penalty of thirty cycles results for a single branch delay slot architecture. Adding the additional 30 cycles to the base 5970 cycles results in a grand total of 6000 cycles to execute control block 2.

Sometimes an instruction does not complete execution before the last instruction due to long instruction latency. In this case, the execution time due to the long latency instruction is also computed within the control block execution time. This is the worst-case estimate of the control block completion time. By adding the completion time for all of the control blocks within the benchmark, an estimate of the total benchmark execution time is obtained.

Estimating execution time with the branch profile can be quick since a trace is collected only once. Since simulation is based on the branch profile, an execution time can be estimated even when the instruction sequence and characteristics change. For example, a nonexisting multiply-add instruction can be simulated by collecting the branch trace first using separate multiply and add instructions, and then modifying the instruction pair into a single multiply-add instruction during the cycle time estimate. The drawback with estimating an execution time with the branch profile is that dynamically variable information such as cache misses cannot be easily integrated. A more flexible trace simulator which can handle dynamically variable information is the subject of the next section.

3.3 Estimating Execution Cycles Using a Simulator

A more detailed processor simulation environment has been developed within the IMPACT project. The simulator is driven by instruction traces and can model in detail different pipeline stages, branch prediction strategies, and memory subsystems. The subject of this section, however, is not the details of the simulator itself, but the process of simulating an architecture with new features. Attention is focussed on the instruction template modification and the benchmark behavior estimation.

Instruction trace and memory trace generation have already been discussed in Section 3.1.2. Two rules must be enforced during instruction template construction and trace collection: 1) the sequences of branches must remain in their original position, and 2) the sequences of memory instructions must remain in their original position. This allows the traced branch to find its basic block and the associated instructions, and the memory instruction to find its dynamic address. As far as the other instructions are concerned, any addition or change to the instruction template will reflect only on the behavior of the simulator while leaving the benchmark executable unchanged. Therefore, by altering the instruction template, different instruction sets and sequences can be simulated while using the existing machine's executable module.

In this thesis, architecture enhancement requires adding new instructions into the instruction set and altering the processor behavior. Existing machines do not recognize the new instructions; the new instructions are recognized only by the simulator. To generate traces for the architecture with new instructions, the executable module must be composed of existing instructions while feeding the simulator with new instructions. This is accomplished by capturing the new architecture's instruction sequence into the instruction template, while the compiler generates the executable module on the existing machine using emulation code sequences. The processor simulator recognizes the new instructions and acts accordingly. To simulate new architecture features, the following steps are taken:

(1) Compile a benchmark for the new architecture using the existing machine's instruction set. Emulate nonexisting features with existing instructions and predefined function calls. Instructions which should not appear in the new architecture's trace are marked. The compiler generates a probed executable module.

(a)

```
L1:
    mov r1, array_A
    mul r2, size, 4
    add r2, r2, r1
L2:
    st  mem(r1),0
    add r1, r1, 4
    bne r1, r2, L2
```

(b)

```
L1:
    init array_A, size
L2:
```

Figure 3.5 Example for executed instructions and the instruction template. (a) Instruction sequence to be executed on an existing machine, and (b) instruction template generated for the new architecture.

(2) The compiler replaces the emulating instructions with new architecture instructions. The resulting instruction sequence is saved into the instruction template. Instructions which should not appear in the new architecture's trace are marked.

(3) Execute the benchmark to collect the branch trace. Branch trace indexes into the instruction template to generate the instruction trace. Unwanted instructions are removed from the trace for the new architecture. Finally, the trace is fed into the processor simulator.

Figure 3.5 provides a simple example. Assuming that the new architecture feature is the init instruction which has the format *init begin_address, zeros_to_fill*. The code segment in Figure 3.5(a) is generated for the benchmark to execute correctly on an existing architecture. The array address and the size are moved into two registers. The loop is traversed until all required elements are zeroed. During simulation, however, when the branch to L1 is taken, the simulator will see only a single init instruction since the template for basic blocks L1 and L2 contains only the init instruction (Figure 3.5). Note that in the instruction template, basic block L2 contains no instructions. The executable may stay in the L2 loop and generate traces. However, since no instruction is found within the template for L2, no instruction is fed into the simulator.

CHAPTER 4

PRELOAD REGISTER UPDATE

By exploiting fine-grained parallelism, superscalar and VLIW processors can potentially increase the performance of future systems. However, these systems may have a long access delay to their first-level memory which can severely restrict the performance of superscalar processors. The performance of superscalar processors, however, is more sensitive to data load latency than their single instruction issue predecessors. A superscalar processor can lose over 30% of its performance when the latency for a data load is increased from 1 to 2 cycles [28].[1] The fact that the performance decreases as the load latency increases indicates that loads are often on the program critical path. One important reason why loads appear on the critical path is that their movement is constrained by stores when there is insufficient memory dependence information available at compile time.

Conventional movement of load instructions is limited by memory dependence analysis. Consider the scheduled code segments in Figure 4.1 for a machine that can issue 2 instructions per cycle with a load latency of 2 cycles. Inconclusive data dependence analysis results prohibit the movement of loads above the stores in Figure 4.1(a). This leads to an empty cycle in the schedule. However, if the loads are determined to be independent of the stores, a more efficient schedule is obtained as shown in Figure 4.1(b). This problem compounds as the processor issues more instructions per cycle, since each cycle in which the processor has to wait for memory references becomes more significant to the overall execution time.

[1] Currently, many commercial processors have a load latency of 2 or more cycles.

(a) Inconclusive data dependence.

```
cycle 1  store  store
cycle 2  load1  load2
cycle 3
cycle 4  use1   use2
```

(b) Conclusive data dependence.

```
cycle 1  load1  load2
cycle 2  store  store
cycle 3  use1   use2
```

Figure 4.1 Problem with data dependence.

In this chapter, a hardware scheme which allows the compiler to perform aggressive scheduling in the presence of inconclusive data dependence analysis results is discussed. This mechanism is referred to as *preload register update*. In Section 4.1, a description of the full design is presented followed by a subset design which incurs less hardware cost. A compiler which takes advantage of preload register update is described in Section 4.2. In Section 4.3, the effectiveness of preload register update is evaluated for a set of nonnumeric benchmarks. Section 4.4 gives the concluding remarks for this chapter.

4.1 Implementing Preload Register Update

The main purpose of preload register update is to provide support for the compiler to boost a memory load above a memory store when their dependence state is not certain. In this section, we discuss the details of one possible implementation of preload register update. The design details will undergo minor modifications as the compiler provides different levels of support. Our compiler support overview will be discussed in Section 4.2.

4.1.1 Overview of the full-scale design

When a load is moved above a store and their dependence relation is uncertain, the load becomes a *preload*. A coherence mechanism must be used to update the preload destination

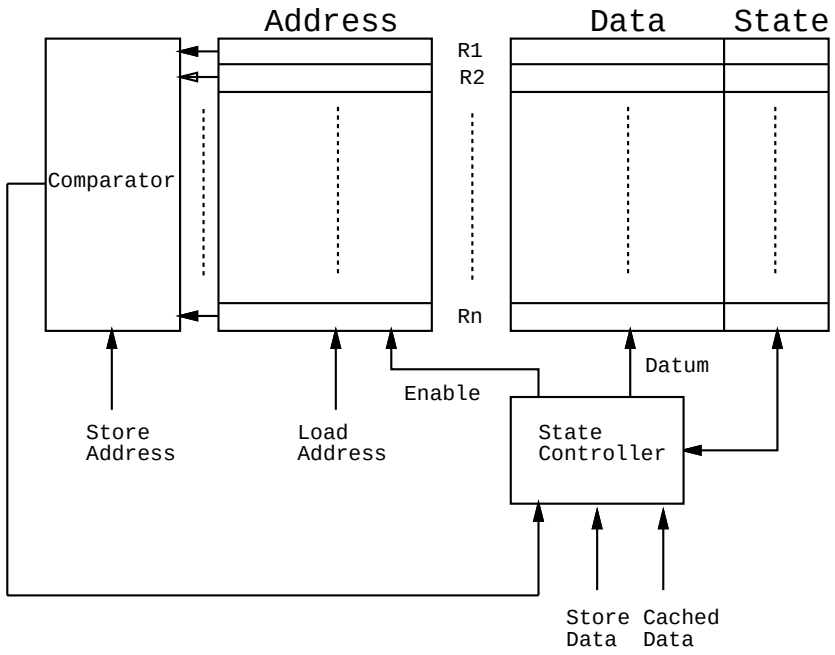


Figure 4.2 Overview of preload register update.

register if the preload and the store reference the same memory location. Figure 4.2 provides an overview of the coherence mechanism. For each register data entry, an address register entry is added. Thus, if we have n general purpose registers, n address registers are added. The purpose of these new registers is to store the addresses of preloads. When a store instruction is executed, the store address is compared against all preload addresses in the address registers. When the addresses match, the stored value is forwarded to the corresponding data register entry for an update. Since there are multiple address registers, a fully associative comparison of the store address and the individual preload addresses must be made. A commit instruction is inserted at the original position of the load (we will discuss the implementation alternatives of this commit instruction in Section 4.1.3). The coherence mechanism will continue to operate until a commit instruction is executed or the register entry is redefined by a normal instruction.

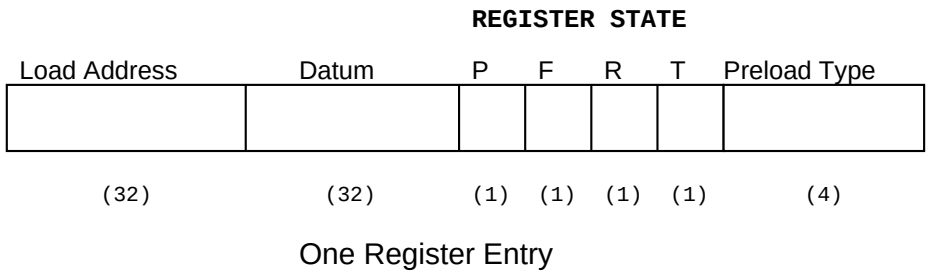


Figure 4.3 Register states implementation detail.

To distinguish between a normal and a preloaded datum, several bits to represent the state of each register are required. The state bits associated with each data register entry are presented in Figure 4.3. The opcode type of the preload is encoded and saved in the preload type field. This is used for data alignment and the masking of a forwarded datum when a store type is different from that of a preload (e.g., preloading a character versus storing an integer). When a preload instruction is executed at run time, the associated preload bit (P) is set for its destination register. The preload state is reset by the corresponding commit instruction, which turns off the coherence mechanism for the register. The ready bit (R), which is similar to the ready bit required by an interlocking mechanism, is set to 0 while the register content is being generated or accessed. If the preloaded address is an I/O port defined by the memory management unit, the freeze bit (F) is set so that the load can be retried at the time of the use. The preload to the I/O port is therefore aborted. We can delay the trap caused by a preload by setting the trap bit (T). If the preloaded value is not used, the trap can be ignored. The detection of exceptions for optimized and scheduled code is discussed in [29]. In this chapter, we focus on the use of the preload register to improve the overall program performance.

4.1.2 Implementation timing and pipeline stages

The stages of the pipeline model are illustrated by an example in Figure 4.4. We wish to demonstrate two issues with this example: updating register content and forwarding data if a preload data is used immediately after a preload/store address match. If a preload address matches the store address, the preload register content is corrected at the write back stage of the store. The computation result writes back and the preload content update due to a matching store address is prioritized according to the instruction execution sequentiality. When more than one preload address matches the store address, the pipeline is frozen until all of the register values are updated.[2] To allow the preload register to be used right after the last bypassed store, the datum for the preload register update is also forwarded to the execution unit that uses the preload. [3] The register file has a direct path from the write back port to the read port to allow such forwarding.

We now concentrate on the P, F, and R bits of the register state. The register states are explained in Table 4.1. The associated state diagram is presented in Figure 4.5 while the possible inputs used in the state diagram are given in Table 4.2. All states with an input of 1 (instruction is a preload) or 9 (register is defined by an instruction other than a preload) will go to states $PF\overline{R}$ and $\overline{PFR}$, respectively. To make the hardware simple, all register data update from redefinitions can proceed only if the register is in its ready state (R=1). Thus, the proposed method is compatible with a processor with a simple interlocking mechanism similar to that used in CRAY-1 [31]. If the freeze state is immediately known at the time of the

[2]This allows us to avoid the complexity of register file implementation for multiple corrections. This trade-off is reasonable because we expect the occurrence of this situation to be rare.

[3]Output and antidependences are handled by register renaming at the decode stage[30]. In this case, the coherence mechanism operates on the physical registers.

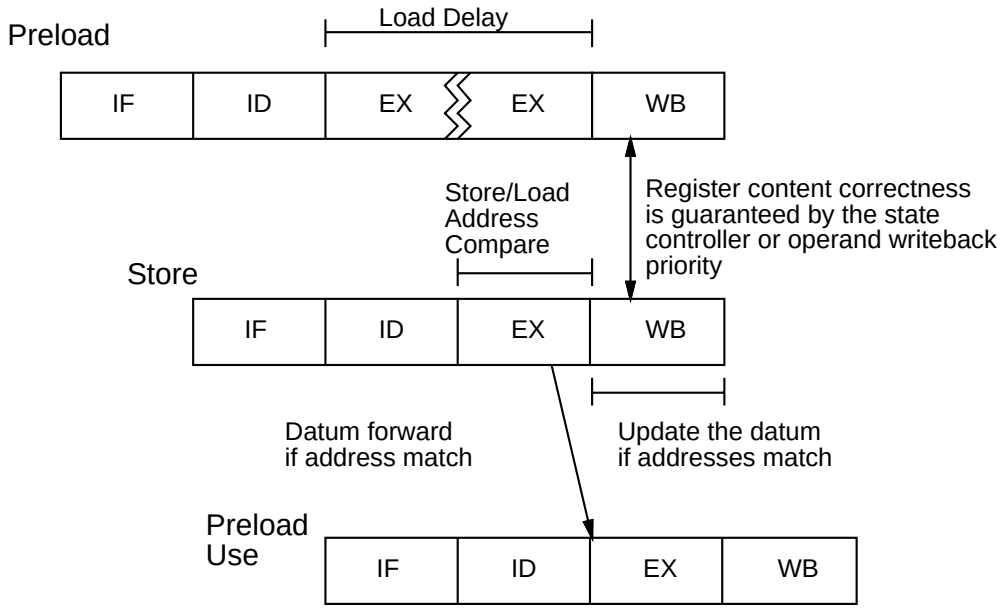


Figure 4.4 Example pipeline stage and instruction relation assuming load latency of 2.

Table 4.1 Explanation of register states.

Register State			*explanation*
P	*F*	*R*	
0	0	0	Register redefined by a nonpreload instruction, not ready
0	0	1	Register contains ready datum, coherence off
0	1	0	State not used
0	1	1	State not used
1	0	0	Preload register, normal datum, not ready
1	0	1	Preload register, normal datum, register ready
1	1	0	Preload register, normal or frozen state unclear, not ready
1	1	1	Preload register, frozen datum

Table 4.2 Input types to the preload register controller.

Type	*Explanation*
1	Instruction is a preload
2	Preload needs to be frozen
3	Preload does not need to be frozen
4	Store address matches the preload address
5	Store address does not match preload address
6	Preload datum is delivered
7	Nonpreload datum is delivered
8	Preload datum used, commit
9	Register is defined by an instruction other than a preload

preload, we can eliminate the 110 state totally and proceed to state 100 or 111 depending upon the freeze status. Although the TLB access is fast in most processors, it still requires some lag time before the result is available; therefore, state 110 is included. State 001 is the initial state of all registers.

4.1.3 Committing preload data

After the execution of all of the stores that were bypassed by the preload, the coherence mechanism is no longer needed for the destination register of the preload instruction. In fact, none of the subsequent stores should be allowed to modify the register. Therefore, a method to *commit* the preload is required to turn off the coherence mechanism at this point of the execution.

A commit instruction can be implemented in two possible ways. First, it can be added to an existing instruction set. This opcode would only have one operand, which is the register number of the preload destination register. The execution of the commit instruction turns off the coherence mechanism, or retries the preload if the freeze bit is set. For the second option, the use of the preload destination register implies a commit instruction. As described

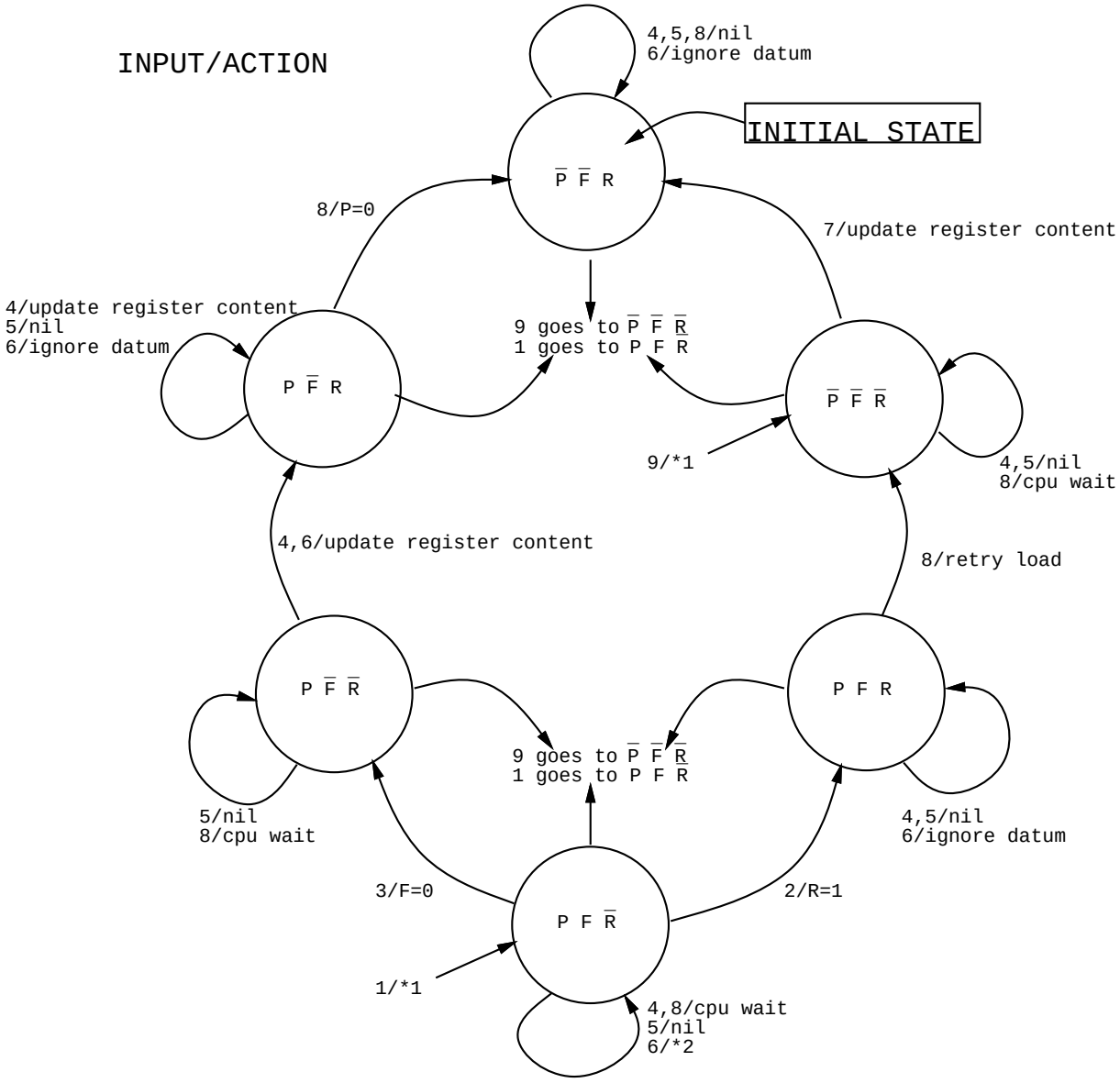


Figure 4.5 State diagram of preload register controller.

in Section 4.2, the compiler does not move the instruction which uses the preload data above or below any stores that may conflict with the preload. Therefore, the execution of the use signals the ending of the coherence mechanism. If a use is not available, we can create an artificial use of the register by performing a *move* to a register hard-wired to 0 (such as R0 in the MIPS R2000 [32]) or to itself. The two alternatives both have their advantages and disadvantages. They vary in hardware complexity, compiler complexity, and execution overhead. From our experience, the explicit commit instruction incurs much more overhead, and is not effective for low issue rate machines. Therefore, we will concentrate on the second model.

4.1.4 An example of preload register update operation

We illustrate preload register update with an example. Figure 4.6(a) shows a load/store pair whose dependence cannot be resolved at compile time. With preload register update, the load is moved above the store with the condition that the use by *op3* remains between the two stores (Figure 4.6(b)). The execution of the preload instruction changes P of $R3$ to 1 to indicate that $R3$ contains a preloaded datum, and memory coherence must be maintained for $R3$. When the store is executed, the coherence mechanism checks the store address against the address field of $R3$, and finds that they are the same (Figure 4.6(c)). Therefore, the data field of $R3$ is updated with the stored value. When the ALU instruction is executed, P of $R3$ is set to 0, thereby turning off the memory coherence for this register.

4.1.5 Subset design of preload register update

At this point, one may question the viability of the full-scale design when the number of address registers increases to a large value. This subsection presents a subset design of preload register update, which incurs lower cost for register files of all sizes.

(a) Original code segment

```
store mem(R1) <- R2
load  R3 <- mem(R4)
alu   R4 <- R3 + 99
store mem(R4) <- R1
alu   R2 <- R3 + R2
```

(b) Code segment after preloading

```
(op1)  preload R3 <- mem(R4)
(op2)  store   mem(R1) <- R2
(op3)  alu     R4 <- R3 + 99
(op4)  store   mem(R4) <- R1
(op5)  alu     R2 <- R3 + R2
```

(c) Sample execution when load and store addresses conflict

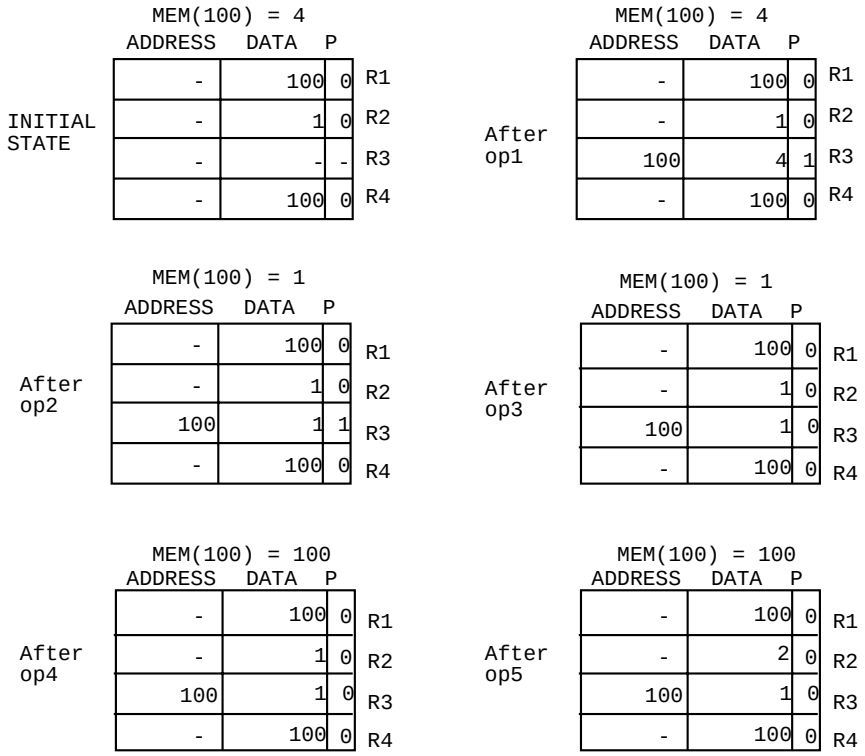


Figure 4.6 An example of preload register update.

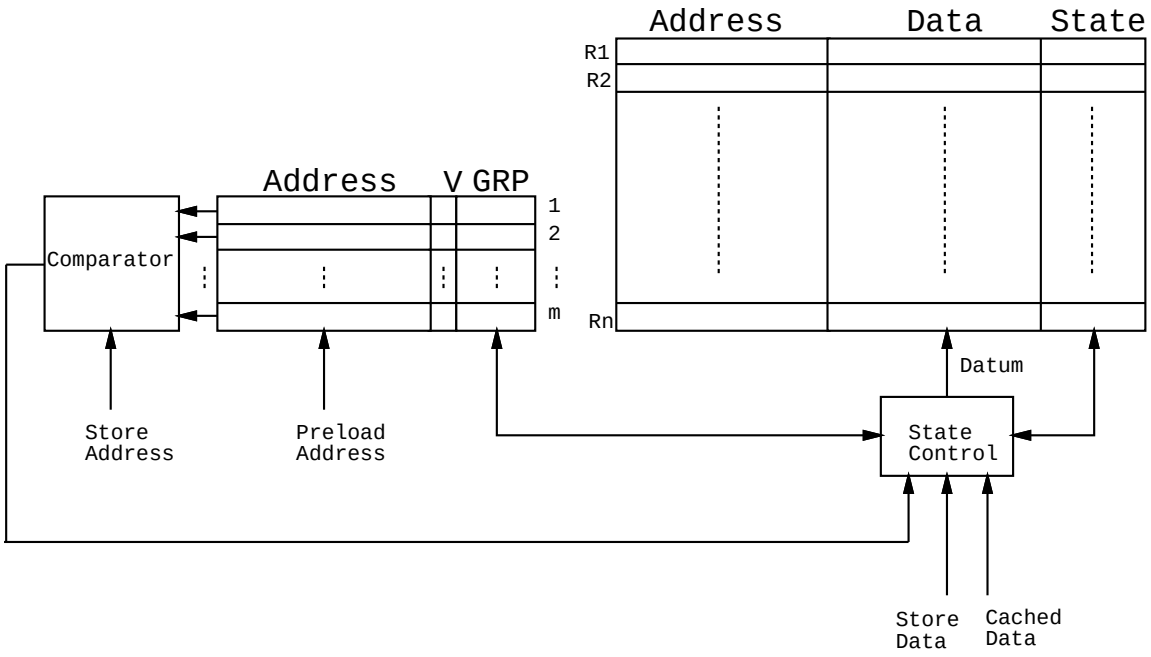


Figure 4.7 A subset design of preload register update.

Basically, the subset design is similar to the full-scale design, except that the number of the fully associative address compares is reduced to m (where $m < n$). That is, there are m address registers each with a general-purpose register pointer (GRP) field and a valid bit (V) added (see Figure 4.7). All other state bits for the general-purpose registers remain unchanged. The purpose of the GRP is to associate an entry of the address register (from here on, the set of m address registers will be referred to as address registers) to an arbitrary general-purpose register entry. This way, all general purpose registers can become a preload register, but only m of them can be active for memory coherence at the same time. The V bit indicates whether the address entry contains a valid address for memory coherence. If the V bit is 1, the register pointed to by GRP has to be kept coherent for all subsequent memory stores. If there is more than one preload address entry which matches the store address, the pipeline is frozen and all matching registers are serially updated.

(a) Original code segment

```
store (R1) <- R2
load  R3 <- (R2)
load  R4 <- (R4)
```

(b) Code segment after preloading

```
(op1)  preload R3 <- (R2)
(op2)  preload R4 <- (R4)
(op3)  store   (R1) <- R2
(op4)  commit  R3
(op5)  commit  R4
```

(c) An example of preload overflow

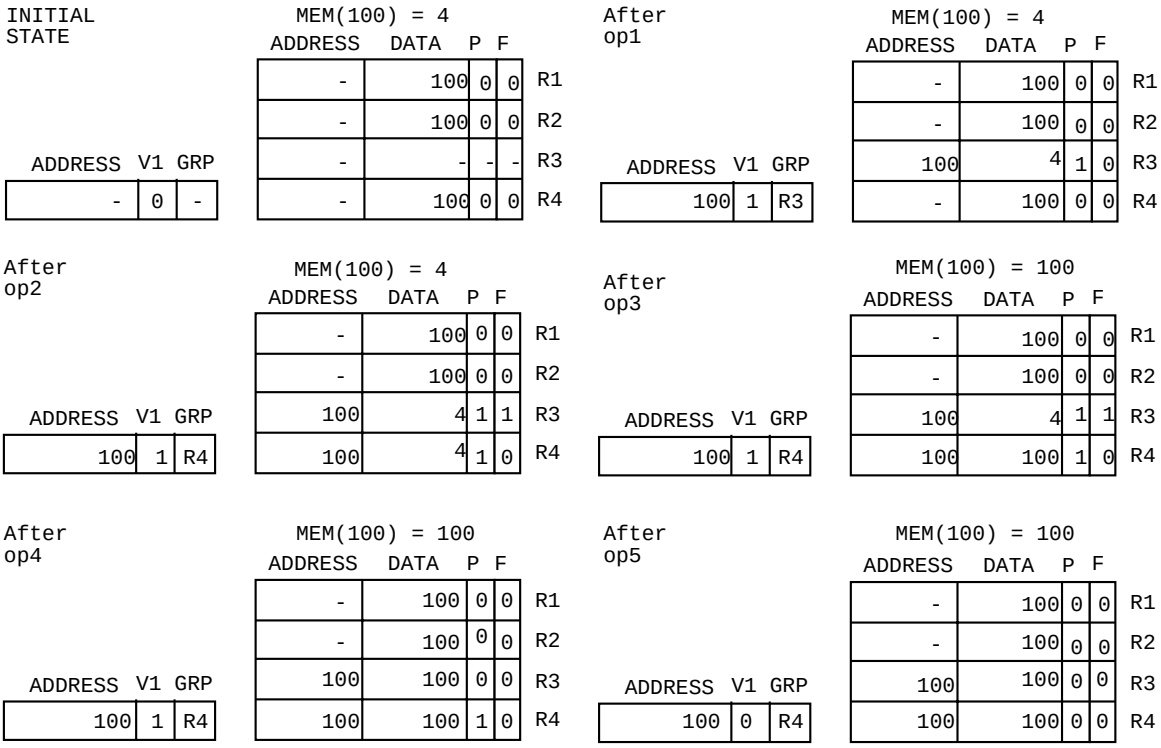


Figure 4.8 An example for preload in subset design.

At run time, a preload can occupy any of the address register entries. When the number of preloads exceeds the number of available address registers, an address register entry is replaced to allow for the new preload instruction. The replacement strategy can be LRU, FIFO, or any other desired replacement policy. However, care should be taken to prevent stale preload addresses from occupying useful address registers due to incorrect compile time branch prediction. At the time of replacing an address register entry, the F bit of the general-purpose register pointed to by the GRP is set to 1. This will cause a retry of the load when the register is used.

To illustrate the operation of the subset design, an example is provided in Figure 4.8 with one address register available. When *op2* finishes executing, the data content of the first preload is no longer kept coherent, and the F bit is set to 1. At the time to commit R3, we retry the load to memory location 100 and obtain the correct datum. All register entries are in the normal data state after *op5* finishes execution.

4.2 Compiler Support for Preload Register Update

In this section, we focus on code scheduling, which is the most important aspect of the compiler support for preload register update. The scheduling support discussed in this paper is based on the superblock structure [28]; however, it can be easily generalized to other structures. A superblock or extended basic block is a block of sequential instructions in which control can enter only from the top but may leave from one or more exit points.

4.2.1 Preload scheduling

To perform superblock scheduling, a dependence graph is constructed for each superblock. The dependence graph includes flow, output, anti, and control dependences between instruc-

tions. In addition, memory dependence arcs exist between all load/store, store/load, and store/store pairs unless the compiler can determine that their respective addresses are always different. With the dependence graph in place, a list scheduling algorithm is used to derive the schedule for each superblock.

To take advantage of preload register update, the dependence graph construction phase has to be modified. Several terms are used to explain the changes. When the memory dependence relation between two memory instructions is uncertain, the dependence is termed *ambiguous*. The Closest Ambiguous Store Before (CASB) of a memory instruction is defined as the first ambiguous store above the memory instruction. The Closest Ambiguous Store After (CASA) of a memory instruction is defined as the first ambiguous store after the memory instruction. The basic block in which a preload originated is called the *home basic block* of the preload.

If a load instruction is not indirectly flow dependent upon another load instruction in the superblock, then it is marked as a potential preload. For potential preloads, all memory dependences on all preceding stores within the superblock are removed. No use of the preload destination register can be moved above the CASB. We stipulate that at least one instruction which uses the result of the preload must remain within the home basic block of the preload. Also, this use is marked as the commit instruction and must be scheduled before the CASA. If a use is not available in the home basic block, a commit instruction is inserted in the home basic block. Note that preloads may be moved above branches during superblock scheduling. In this thesis, the general code percolation model [28] is assumed, and nontrapping hardware [33] is used to suppress the exceptions caused by these preloads.

An example dependence graph for the code segment in Figure 4.9(a) is shown in Figure 4.10(a). We assume a load latency of 2 cycles and a latency of 1 cycle for all other in-

(a) Original code segment

```
t1:    ALU
t2:    BRANCH
t3:    STORE
t4:    LOAD
t5:
t6:    ALU
t7:    STORE
```

(b) Code segment after preloading

```
t1:    ALU   LOAD
t2:    BRANCH
t3:    STORE
t4:    ALU
t5:    STORE
```

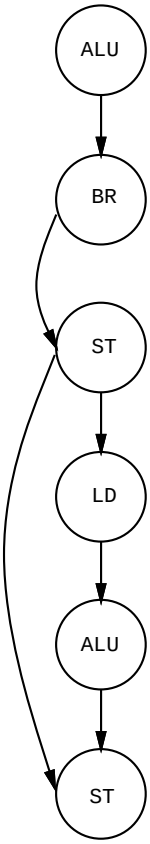
Figure 4.9 Code scheduling and execution cycles.

structions for this example. To take advantage of the preload register update support, the dependence from the store to the load is removed. A new dependence constraint now exists from the store to the second ALU instruction. The updated dependence graph is shown in Figure 4.10(b). By allowing the load to bypass the first store, the second ALU instruction can be scheduled earlier, thereby shortening the program critical path for a high issue rate processor. The resulting schedule is shown in Figure 4.9(b). The total execution time drops from 7 to 5 cycles. Note that if the load latency is 3 cycles instead of 2 cycles, the original code segment in Figure 4.9(a) would have taken 8 cycles to execute. However, the execution time of the code segment with preloading in Figure 4.9(b) remains at 5 cycles.

4.2.2 Data register issues

Since memory coherence will not operate properly if the register content is saved somewhere else (e.g., on the stack), there are certain restrictions placed on the handling of preload destination registers. Without interprocedural register allocation, preloads cannot be moved above a function call. Also, the register allocation algorithm has to be modified so that the preload destination registers tend not to be spilled before they are committed. This is accomplished by

(a) Original dependence graph



(b) Dependence graph after memory bypass

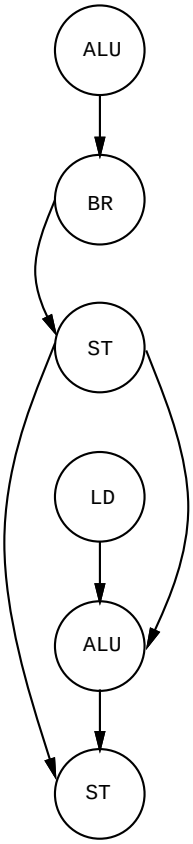


Figure 4.10 An example dependence graph.

increasing the live range weight of the preload destination registers before their corresponding commit instruction. Thus, spilling will be unlikely for the register in that section of the code. If spilling does occur, the data register, the address register, and the register state all have to be saved to the stack. [4] Whenever a spilled preload destination register is filled from the stack, its F bit is set to 1. Thus, at the time of the use, the register value is reloaded from the data cache, thereby obtaining the most recent value.

4.3 Experimental Evaluation of Preload Register Update

Compiler support for preload register update has been implemented in the IMPACT-I compiler. In this section, experimental results on the effectiveness of preload register update are reported for the thirteen nonnumeric C benchmarks listed in Table 4.3. The profile information is used to identify superblocks in the benchmark programs. All benchmark programs are profiled with several different inputs.

4.3.1 Evaluation methodology

To evaluate the performance of preload register update, each benchmark program is executed using one input different from those with which it was originally profiled. During execution, statistics are gathered for all branch instructions. Later, the benchmark is statically scheduled with the preload register update support. The execution cycles for the instruction issue rates of 2, 4, and 8 are determined by branch profile as discussed in Section 3.2. One branch delay slot that consists of N instructions for an N-issue processor is automatically allocated for

[4] In fact, only either the data register or the address register has to be saved onto the stack depending on the preload status (P bit) of each register. If the register is marked as a preload register, then only the address register has to be saved, otherwise, the data register is saved.

Table 4.3 Benchmarks.

Benchmark	*Size*	*Benchmark Description*
cccp	4787	GNU C preprocessor
cmp	141	compare files
compress	1514	compress files
eqn	2569	format math formulas for troff
eqntott	3461	Boolean equation minimization
espresso	6722	truth table minimization
grep	464	string search
lex	3316	lexical analyzer generator
qsort	136	quick sort
tbl	2817	format tables for troff
wc	120	word count
xlisp	7747	lisp interpreter
yacc	2303	parser generator

each predicted-taken branch instruction. The function units are pipelined and uniform for all issue rates except stores, which are restricted to one per cycle due to the difficulties involved in designing the associative search and forwarding logic to handle multiple stores per cycle. Therefore, for an N-issue machine, N loads can be issued in the same cycle, but at most one store along with N-1 other instructions can be issued in the same cycle. The instruction latencies for different function units are listed in Table 4.4. Load latency can vary due to different cache sizes and physical distances (e.g., on-chip or off-chip). Thus, the load latency is varied from 1 to 4 cycles for the experiments. The processor includes a 64-entry integer register bank and a 32-entry floating-point register bank; the full-scale design of the preload register update is used.

4.3.2 Base architecture performance

Figures 4.11 to 4.13 shows the benchmark speedup for issue rates of 2, 4, and 8 with load delays of one to four. This is the base architecture support using the general code percolation

Table 4.4 Instruction latencies.

INT function	*latency*	*FP function*	*latency*
ALU	1	ALU	3
barrel shifter	1	conversion	3
multiply	3	multiply	4
divide	10	divide	10
load	varies	load (1 word)	varies
preload	varies	preload (1 word)	varies
store	1	store	1

model without preload register update support. It is intended to show the problem of an increase in load delay on superscalar performance. The x-axis shows the benchmark speedup, and the base is a single issue processor with the same load delay. The y-axis lists the name of each benchmark. The number beside each point on the graph indicates the load latency that is under consideration.

Increase in load delay has more negative effect for issue 4 and issue 8 processors than two-issue processors. Also, the performance degradation is often greater when going from a load latency of 1 to a load latency of 2. This is especially true for the issue 8 processor in Figure 4.13. It is, therefore, important for higher-issue rate processors to have small load latency. Preload register update support attempts to provide the illusion of small load latency for high-issue processors with large load latency.

4.3.3 Performance evaluation of preload register update

The full-scale design of preload register update is evaluated here in terms of execution speedup with an ideal cache. For each individual issue rate and load delay, we take the ratio of the execution cycle of the benchmark without preload register update support to the execution cycle of the code with preload register update support to calculate the speedup. We can

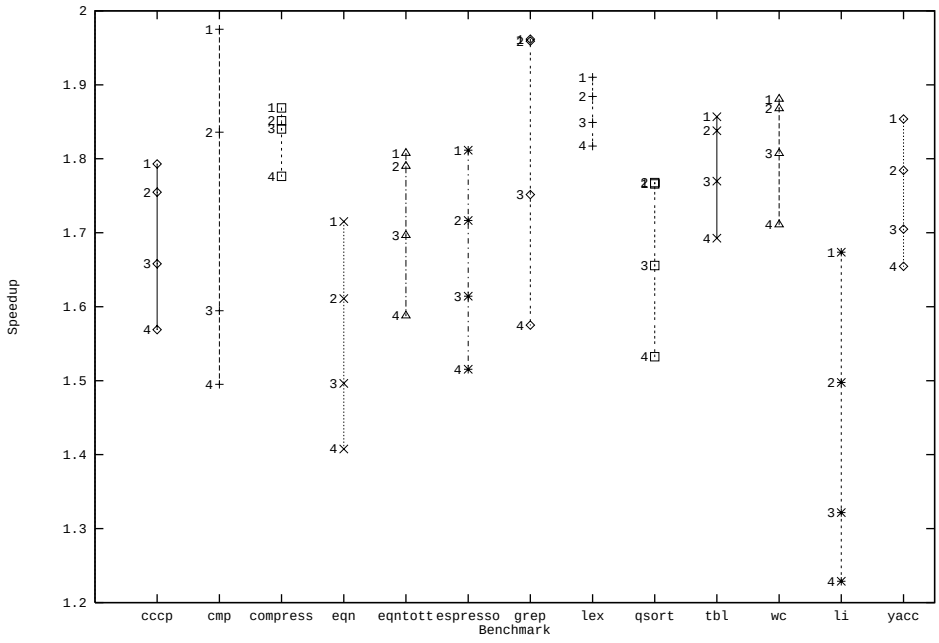


Figure 4.11 Speedup with varying load latency, issue 2.

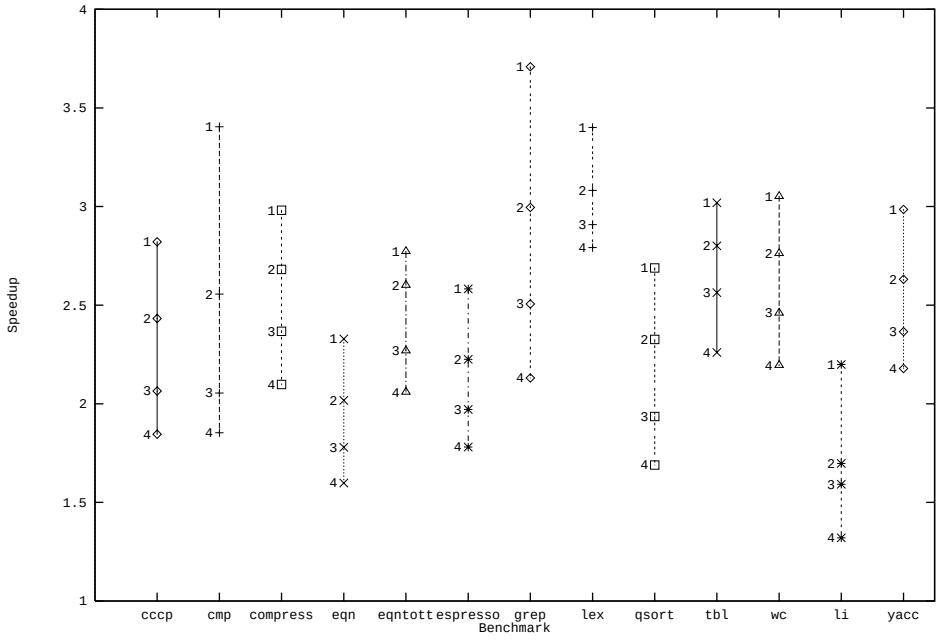


Figure 4.12 Speedup with varying load latency, issue 4.

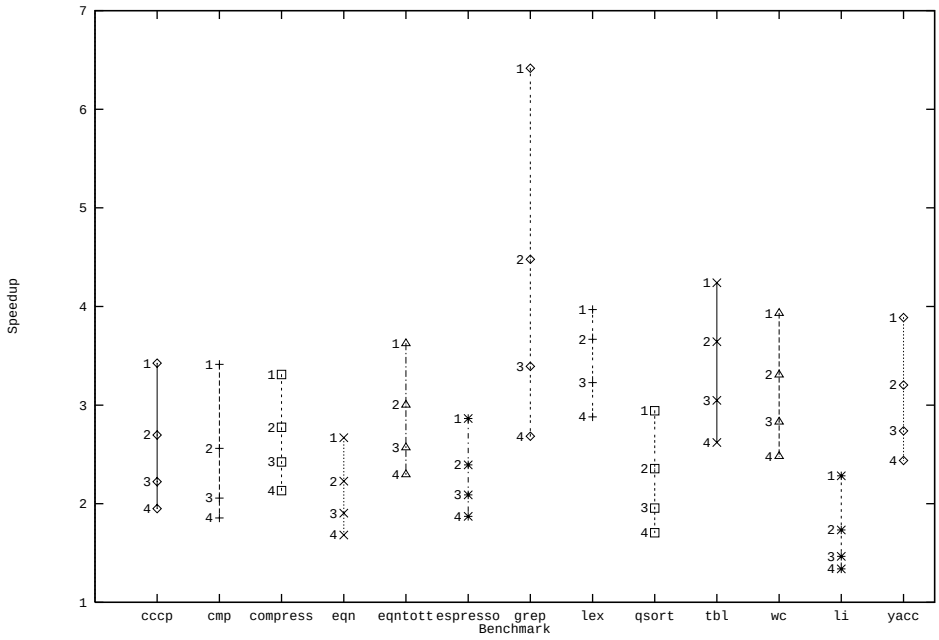


Figure 4.13 Speedup with varying load latency, issue 8.

characterize the thirteen benchmark programs into two groups. The first group pertains to the benchmarks that obtained a noticeable performance improvement with preload register update. The second group consists of the benchmarks which did not benefit significantly from preload register update. Figures 4.14 , 4.15, and 4.16 show the speedup achieved by superscalar processors over the base architecture with and without preload register update. These figures are in the same format as Figure 4.13. We will examine the two groups separately and explain the difference.

The benchmarks in the first group are able to tolerate the increased load latency better with preload register update. The benchmarks in this category are *cccp, cmp, eqn, espresso, grep, qsort, wc*, and *xlisp*. Closer examination of the benchmark programs reveals that the scheduling is limited by inconclusive memory dependence analysis results. As a result, the

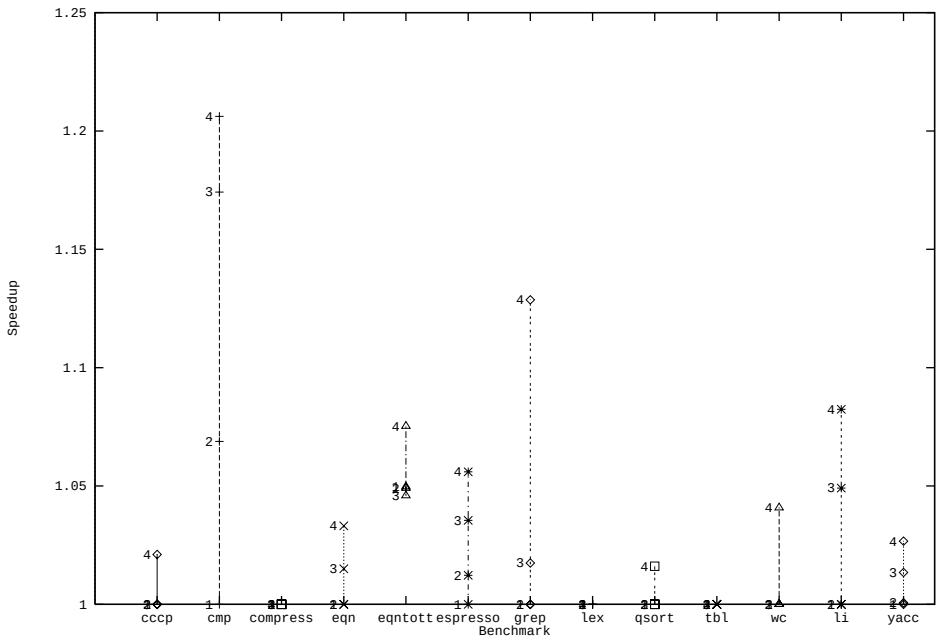


Figure 4.14 Speedup for benchmarks, issue 2.

extra freedom to reorder memory instructions provided by preload register update leads to substantial performance improvement.

The benchmarks of the second group do not benefit from preload register update in general. The benchmarks in this category are *compress, eqntott, lex, tbl,* and *yacc*. Examining these benchmarks shows that the scheduling of these programs is not restricted by memory dependence analysis. Also, a lack of stores in the frequently executed regions within these benchmarks provides more scheduling freedom for load instructions than do the benchmarks of the first group. These benchmarks are examples in which there are few opportunities for preload register update to further improve performance.

The results shown do not necessarily mean that the Group 2 benchmarks cannot tolerate the increased load latency. For high issue rates, the performance decrease that arises as the load

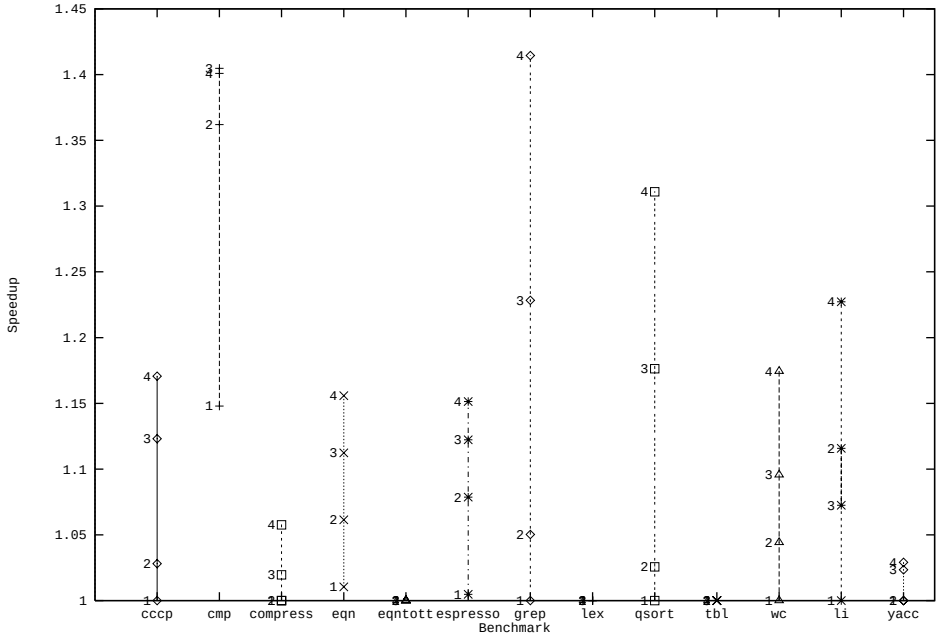


Figure 4.15 Speedup for benchmarks, issue 4.

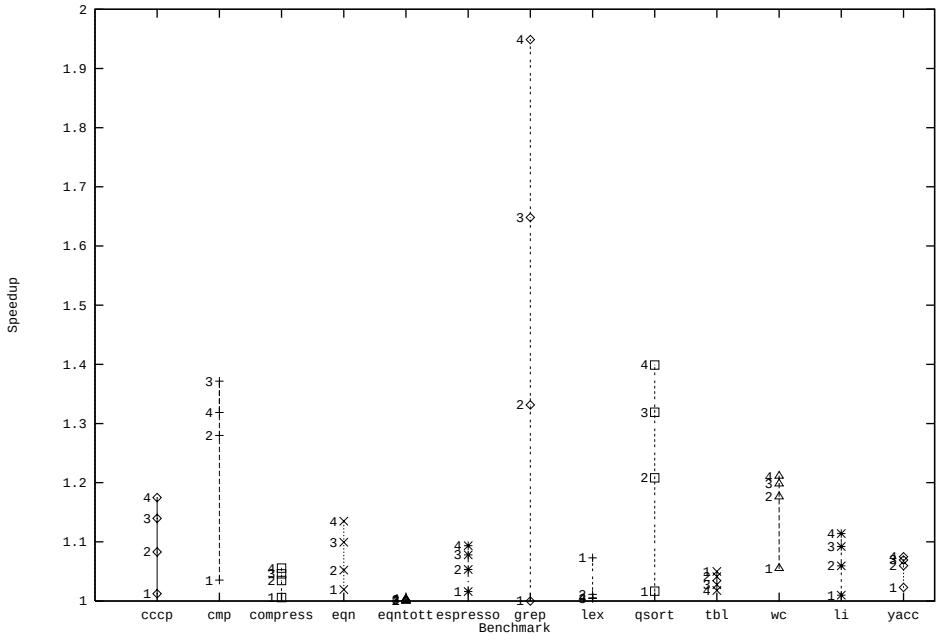


Figure 4.16 Speedup for benchmarks, issue 8.

latency increases is due to the lack of schedulable instructions within the superblock. Further loop unrolling is required to provide sufficient independent instructions to hide the load latency.

Another observation is that preload register update is more effective for processors with longer load latency. Scheduling load instructions early becomes more crucial as load latency increases. Without conclusive memory dependence analysis results, as in the case of first group benchmarks, load instruction cannot be scheduled sufficiently early to hide the load latency. Preload register update removes the memory dependences which restrict the movement of load instructions. Therefore, the schedule allows the processor to better tolerate the increased load latency.

4.4 Summary of Preload Register Update

Preload register update complements compile-time data dependence analysis. Without conclusive data dependence analysis results, conventional compile-time scheduling of memory instructions is restricted by conservative assumptions. Preload register update allows the compiler to move load instructions even in the presence of inconclusive data dependence analysis results. The load destination registers are kept coherent when load instructions are moved above store instructions that reference the same location. We have addressed issues regarding data forwarding, register interlocking, and register coherence within the context of a detailed state diagram and a processor pipeline example. Problems associated with memory mapped I/O ports and registers are resolved with additional register states. Saving and restoring (e.g., register spills) of preload registers are shown to provide correct operation. Lastly, a subset design of preload register update which incurs less cost while maintaining similar functionality is discussed.

A major limitation of preload register update is that dependent instructions of a load may not be scheduled above any ambiguous stores. This is because the execution effect of these dependent instructions cannot be reversed if an ambiguous store writes to the memory location referenced by the load. An extension which overcomes this limitation is the subject of the next chapter.

CHAPTER 5

MEMORY CONFLICT BUFFER

A compile-time code scheduler improves the performance of VLIW and superscalar processors by exposing simultaneously executable instructions to the hardware. To be effective, the scheduler must be able to reorder instructions freely within the scheduling scope. Unfortunately, compile-time code reordering is often inhibited by potential dependences between memory reference instructions. Because memory references often occur on program critical paths, such loss of code reordering opportunities can limit the effectiveness of compile-time code scheduling.

Figure 5.1 shows three schedules with different code reordering constraints. The load and ALU latencies are assumed to be two and one cycles, respectively. Assume that data loaded by load1 (load2) are used by alu1 (alu2) whose result is in turn used by store1 (store2). Further assume that store1 and load2 form an ambiguous reference pair. Figure 5.1(a) shows a total serial code segment by assuming a dependence between store1 and load2. If load2 is allowed to be moved above store1, a better schedule can be derived as shown in Figure 5.1(b). Note that in Figure 5.1(b), the performance improvement is limited by the fact that alu2 cannot be moved above store2. If this constraint can be eliminated to further allow alu2 to move above store 1, then the improved code schedule is shown in Figure 5.1(c).

In Section 5.1, an architectural feature referred to as the *memory conflict buffer* (MCB) is introduced to increase the opportunities of compiler-time code reordering. This is achieved by removing the dependence between a store/load pair when they seldom access the same location. With the MCB, a memory load and its dependent instructions can be moved above any number

(a) No code reordering

```
t1: load1
t2: nop
t3: alu1
t4: store1
t5: load2
t6: nop
t7: alu2
t8: store2
```

(b) Load/store reordering only

```
t1: load1
t2: load2
t3: alu1
t4: store1
t5: alu2
t6: store2
```

(c) No constraint

```
t1: load1  load2
t2: nop    nop
t3: alu1   alu2
t4: store1 store2
```

Figure 5.1 Effect of architectural support on compile-time code scheduling.

of ambiguous stores. Section 5.2 discusses a set of compiler algorithms and its implementation to take advantage of the MCB hardware. The effectiveness of this scheme is evaluated for a set of nonnumeric benchmarks in Section 5.3.

5.1 Architectural Support

When a load is moved above an ambiguous store, the load becomes a *preload*. Instructions that use the preloaded data are also free to move above the ambiguous store. The MCB hardware supports such code reordering by (1) detecting the situation in which the ambiguous reference pair access the same location and (2) invoking a correction code sequence supplied by the compiler to restore the correctness of the program execution. The situation in which a preload and an ambiguous store access the same location will be referred to as a *conflict* between the ambiguous reference pair. When this happens, the computation involving the preload destination register must be redone.

The MCB hardware is physically located close to the on-chip data cache if there is one. Otherwise, it should be located close to the load/store address pads leading to the off-chip data cache. As a separate entity, the MCB support does not alter the CPU register file design. In this

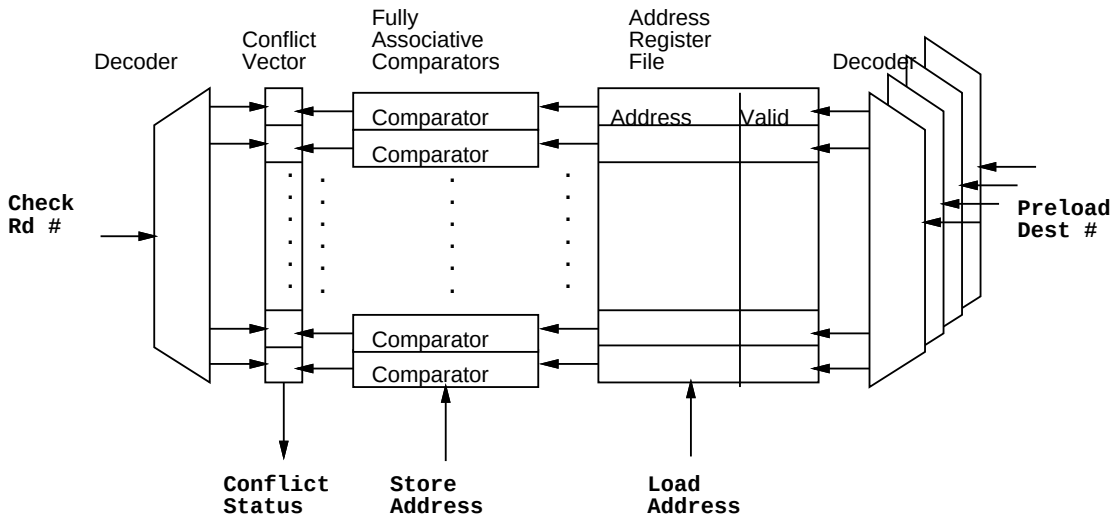


Figure 5.2 Full-scale MCB design.

section, we present two alternative MCB designs to illustrate the cost performance trade-offs involved.

5.1.1 Full-scale MCB design

Figure 5.2 shows a full-scale design of the MCB. One *address register* is provided for every general-purpose register defined in the instruction set architecture. In addition, one conflict bit is associated with each general purpose register to keep track of the occurrences of conflicts. This design is suitable for supporting processors with a moderate number of general-purpose registers. An alternative design for supporting processors with a large number of general-purpose registers will be described in the next section.

Each address register contains a valid bit and a virtual address. When a preload is executed, its destination register number is used to select the corresponding address register. The preload deposits its virtual address into the address register and marks the address register valid. Note that for a processor that supports the execution of multiple preload instructions per cycle, the

address register file must be multiported to allow simultaneous insertion of multiple preload addresses.

When a store instruction is executed, its virtual address is compared against all valid preload addresses in the address register file. This requires fully associative comparison and therefore limits the number of general-purpose registers that can be efficiently supported by this design. If a match occurs in an address register, the corresponding bit in the conflict vector is set. By setting the conflict bit for a general-purpose register, the store signals the need to reload this register from memory and to re-execute its dependent instructions. Note that there can be multiple preloads that match with the same store. Therefore, the comparators are designed to independently set their corresponding conflict bits.

A new conditional branch opcode, *check*, has to be added to the instruction set. The format of the check instruction is as follows:

check Rd, Label

where *Rd* is a general-purpose register number, and *Label* specifies the starting address of the correction code supplied by the compiler. When a check instruction is executed, the conflict bit for Rd is examined. If the conflict bit is set, the processor performs a branch to the correction code marked by *Label*. The correction code provides for re-execution of the preload and its dependent instructions. A branch instruction at the end of the correction code brings the execution back to the instruction immediately after the check. Normal execution resumes from this point.

The conflict bits are reset in two ways. First, a check instruction resets the conflict bit for register *Rd* as a side effect. Second, any preload that deposits a value into a general-purpose

register also resets the corresponding conflict bit. As far as the valid bits are concerned, a check instruction resets the valid bit in *Rd*'s address register.

Whenever a general-purpose register has to be saved to the memory due to context switches, neither the conflict bit nor the address register requires saving. The only requirement is that the hardware set all of the conflict bits when the register contents are restored from memory. This simple scheme causes a performance penalty only when a preload destination register is saved before it is examined by a check instruction. Thus the concern is that extra invocations of the correction code due to context switching will occur. However, from our experience, this overhead is negligible for systems with context switch intervals of more than 100,000 instructions. Issues concerning procedure calls and register spills are discussed in Section 5.2.

Note that only preloads, stores, and checks have to access the address registers and the conflict vector. These accesses are performed in parallel with the data cache access. Thus, the timing constraints for accessing MCB are the same as those for accessing the data cache. Because the MCB support does not alter the design of the general-purpose register file, it is unlikely to affect the processor pipeline timing.

5.1.2 Set-associative MCB design

In this section, we present a set-associative design of MCB in which the number of comparators used does not increase with the number of general-purpose registers. Therefore, the design is suitable for processors with a large number of general-purpose registers. Figure 5.3 illustrates the design.

Each address register in the set-associative design contains a valid bit, a virtual address, and a general-purpose register number. These address registers are organized into S sets of

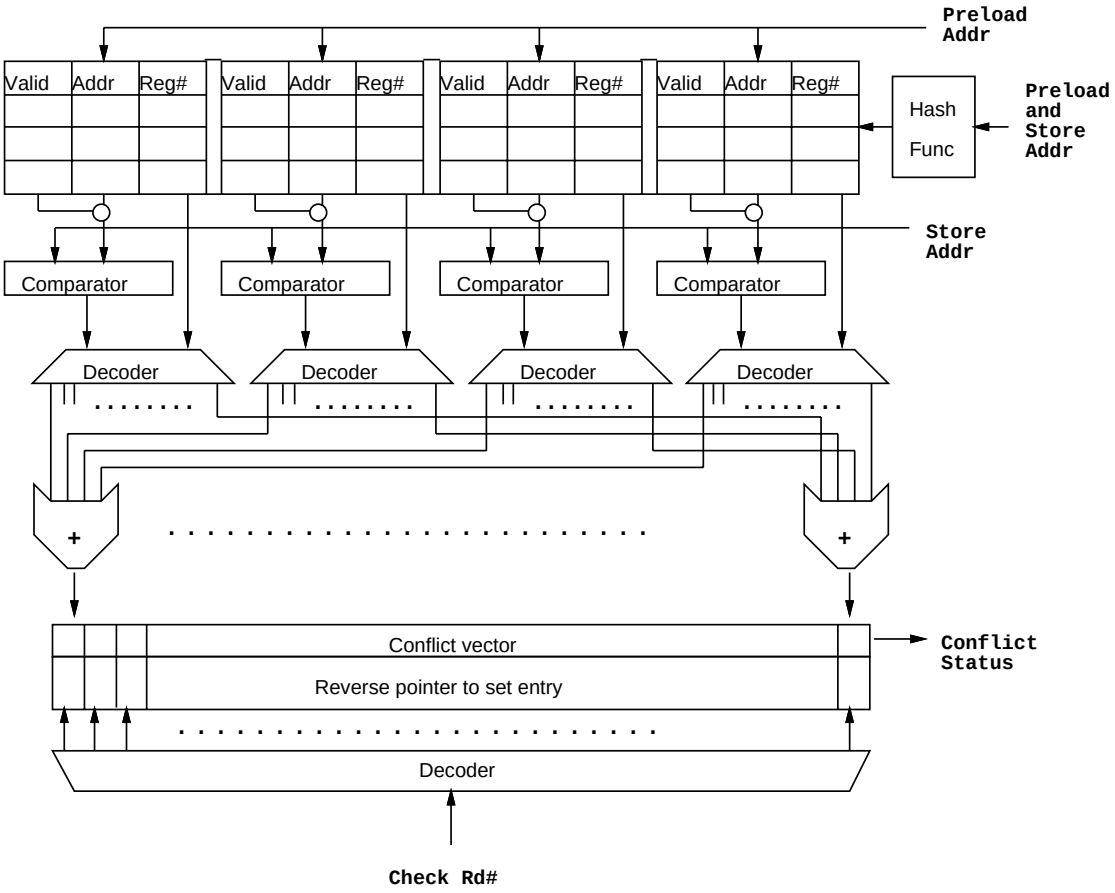


Figure 5.3 A 4-way set-associative MCB design.

N entries each. As in the full-scale MCB design, one conflict bit is still associated with each general-purpose register. When a preload instruction is executed, its virtual address is hashed to select a set. If there is an invalid entry in the set, the entry will be used to host the virtual address and the destination register number of the preload. If all entries are valid, the MCB controller carries out the following replacement policy:

(1) If there is an entry that has been marked for replacement by the approximate LRU algorithm to be described below, the entry is purged to make room for the incoming preload. This rule is used to prevent stale preload addresses from occupying useful MCB space due to incorrect compile-time branch prediction. The general-purpose register number of the replaced entry is used to set its corresponding conflict bit.

(2) If there is no entry found according to the previous rule, no entry is replaced. Rather, the incoming preload is rejected from the MCB. The conflict bit of its destination register is set. This will force the processor to invoke the correction code upon executing the corresponding check instruction. The rationale behind this rule will become clear after we explain the handling of stores.

To support the approximate LRU algorithm, a two-bit age counter is added to each address register. The age counter is initialized to 11 when a preload deposits into the address register. The hardware periodically decrements the age counters of all address registers. Once an age counter reaches 0, it is not further decremented. The decrement of age counters can be done as a background activity in parallel with the normal accesses to the address registers. By making the period of decrementing reasonably large, this method identifies potentially stale address registers when their age counters reach 0.

When a memory store is executed, its address is also hashed to determine the set of address registers to compare with. This set-associative design reduces the size of comparators to the number of entries within a set (N). Typically, N is a small number, e.g., 4, compared to the total number of general-purpose registers. If a conflict is detected, the conflict bit pointed to by the preload destination register number is set, and the valid bit of the corresponding address register is reset. Since there can be up to N conflicts detected for a store, the conflict vector has N write ports. Program execution continues until a check instruction discovers the conflict and invokes the conflict correction code.

Note that the valid bit of an address register is reset immediately after a conflict is detected with the address register. Therefore, if all of the address registers in a selected set are valid, these entries have survived all of the stores they have encountered so far. They will typically have to survive fewer stores than the incoming preload address. Therefore, in rule (2) of the replacement policy, we choose to reject the incoming preload rather than replacing existing valid address registers.

5.1.3 Hash MCB design

In this section, we introduce the hash MCB design in which the address comparisons are eliminated. The hash MCB design provides full MCB functionality except that the address comparison is replaced by a hashing function. This alternative approach requires the same preload and check instructions. However, the MCB is implemented as a direct-mapped buffer, based upon some hashing of the memory address.

Figure 5.4 shows the block diagram of the hash MCB design. When a preload occurs, the address of the preload is hashed to determine the corresponding MCB location. The destination

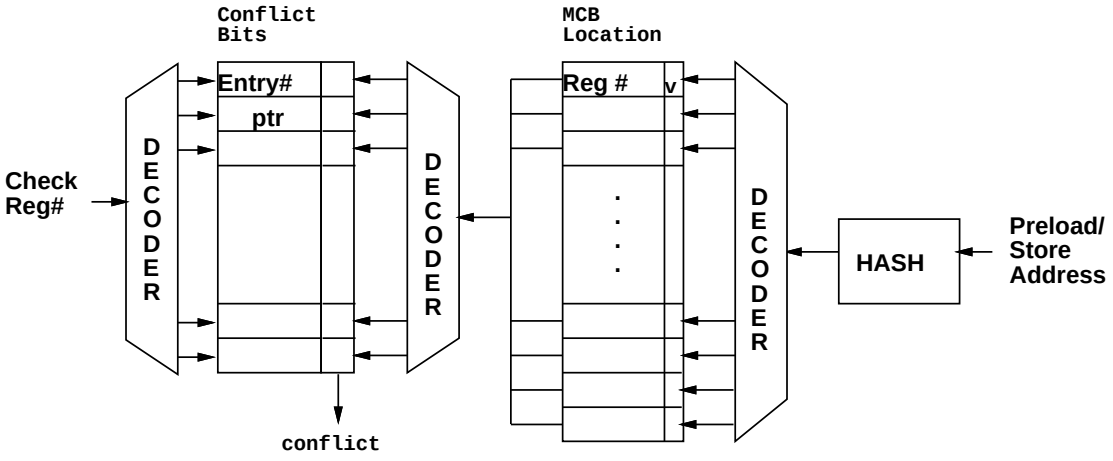


Figure 5.4 Hashing MCB design.

register number of the preload is stored in the MCB and the valid flag is set. Additionally, the pointer of the conflict bit is pointed to the MCB location being accessed and the conflict bit is reset. If the MCB location is originally valid, then the replacement causes the conflict bit corresponding to the replaced register number to be set. The associated pointer is set to null.

When a store occurs, the address of the store is hashed to determine the corresponding MCB location. If the MCB location is invalid, no further action is needed. If the MCB location is valid, then the register number is used to select and set the conflict bit. The associated pointer is set to null. Furthermore, the MCB location is invalidated.

A conflict is generated for a valid MCB entry when 1) it is removed by another preload which hashes to the same entry and 2) a store hashes to the same entry. The conflict is detected by the check instruction. The check instruction examines the valid bit. A conflict bit which is set generates a conflict signal; thus the conflict correction code is executed. The conflict bit is reset at this point. However if the conflict bit is originally not set, the MCB location pointed to by the conflict bit pointer is invalidated to prevent unnecessary conflicts later.

This hash MCB approach will detect all true load/store address conflicts. However, it suffers from false conflicts because addresses are hashed rather than fully compared. There are two types of false conflicts: load/load conflicts and load/store conflicts. All load/load conflicts are false conflicts; these false conflicts result when two loads, possibly to different addresses, hash into the same MCB location. To make room in the MCB and ensure that no true conflicts are missed, we must assume that the load purged from the MCB would have conflicted with a subsequent store. Many load/store conflicts may also be false, because conflict is assumed based upon an address hashing rather than a full comparison. False conflicts cause no problem to the correct execution of the program; however, they impact performance due to the execution of unnecessary correction code. Thus, the hashing MCB approach suffers from some level of performance degradation as compared to the full comparison approach due to false conflicts.

5.1.4 Handling variable data sizes

Many instruction set architectures allow memory references to have byte addressability and variable data sizes. One might argue that these features can potentially complicate the MCB conflict detection mechanism. However, this complication does not usually occur due to the way memory access is implemented in processors. To keep the cache design simple, the processor/cache interface often has logic to perform byte shifting and sign/zero extension. As a result, the cache can be simply accessed with aligned word requests. However, more false load/load conflict will be seen in hashing MCB. Since MCB is physically close to the cache, it can be accessed in the same manner as the cache. The only additional support required is for the CPU to provide a mask about the bytes that are actually accessed in a request. This information is recorded in the address register and is used in detecting conflicts. For a store

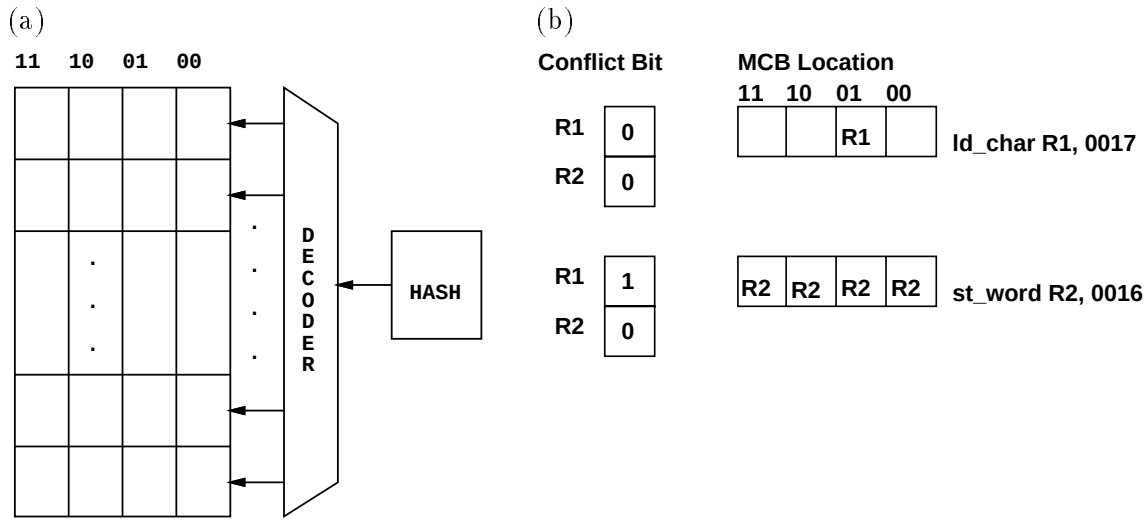


Figure 5.5 (a) MCB extension to differentiate between access sizes and (b) an illustrative example.

to conflict with a preload, the logical AND of the masks must be nonzero and the two-word addresses must match. The mask allows two accesses to different bytes of the same word to be identified as independent of each other.

Providing the access mask, however, does not work as well for the hash MCB design when byte and half-word accesses are supported within the architecture. To ensure that accesses within the same word boundary are hashed into the same entry, lower-order bits of the address must be ignored during address hashing. This will degrade the performance of programs making frequent sequential byte or half-word accesses, since the different lower-order bits do not matter during hashing and the respective addresses will map to the same entry.

Extending the MCB entry in size to allow distinctions between byte, half-word, and word accesses resolves the problem. A block diagram of the proposed extension is shown in Figure 5.5(a). Memory conflict buffer operation with the extended entries is the same as before except that only the subentry is affected by the particular access. For example (in Figure 5.5(b)),

a byte access can modify only one of the four subentries which it is hashed into. For a word access, all four subentries are modified.

5.1.5 Speculative execution

Speculative execution has been used in the literature to refer to executing an instruction before knowing that its execution is required. By moving an instruction above preceding conditional branches, an instruction is executed speculatively. In this thesis, we extend the definition of speculative execution to refer to executing an instruction before knowing that it can be executed correctly. Instructions thus executed will be referred to as speculative instructions. In particular, a preload and its dependent instructions are executed before knowing if the preload will conflict with a store. The execution of these speculative instructions must be corrected if a conflict occurs.

There are two aspects of correcting the execution of speculative instructions. One is to correct the values generated by these instructions. The compiler algorithm described in the Section 5.2 fulfills this requirement through intelligent compile-time renaming and register assignment. The more difficult aspect is to correctly handle exceptions. Since the value preloaded into the register may not be correct, there is a chance that a flow-dependent instruction that uses the preload result may cause an exception. In the example of Figure 5.6, if $r1$ equals $r2$, then the value 7 is loaded into $r3$ in the original code segment. However, the value 0 may be preloaded into $r3$, in which case the divide instruction will cause an exception. Since the exception is due to an incorrect execution sequence, it must be ignored.

A solution is to provide architectural support to suppress the exceptions for speculative instructions [33]. A potential trap causing instruction executed speculatively should be converted

```
(a) Original                    (b) With preload
   store (r1) <- 7                 preload r3 <- (r2)
   load  r3 <- (r2)                div     r4 <- r4,r3
   div   r4 <- r4,r3               store   (r1) <- 7
                                   check   r3, L3
```

Figure 5.6 Example showing the problem of speculative execution.

into the nontrapping version of the instruction. Therefore, the exception caused by the divide instruction in the example above would be ignored. However, the exception should be reported if there is no conflict between the preload and the store. A scheme for the precise detection of exceptions with speculative execution has been proposed and evaluated by Mahlke et al. [29]

5.1.6 A prototype implementation

A gate-level design of the set-associative MCB has been completed, allowing us to estimate transistor count by computing the equivalence in gate count. The design is broken down into five components (Figure 5.3): 1) the hashing function, 2) preload address register storage for the effective addresses and register numbers, 3) the chooser for MCB entry selection, 4) the conflict bit and the reverse entry pointers, and 5) the control logic.

The hashing function, based upon XOR gates, performs permutation-based hashing on the effective address. Including in the hashing function logic is the decoder which selects the appropriate MCB set. The preload address registers store the incoming preload address and preload destination register numbers. The chooser selects the conflict bit location marked by one of the valid entries of preload address registers. The chooser also serves the purpose of reversing the selection process from the conflict bit to one of the preload address registers for invalidation. The conflict bit, one for each physical register, is set if the store address matches the preload address selected. The reverse entry pointer associated with each conflict bit points

Table 5.1 MCB components and transister count.

Component	*Transistor Count*	*Component Characteristics*
Hash function	500	XORs outputing 6 hash address bits 6 to 64 entry decoder
Address register	48700	64 address registers holding 32 bits each for address and 6 bits each for register number
Chooser	1000	Quadruple 2-line-to-1-line data multiplexers
Conflict bit	7700	64 conflict bits with reverse pointers
Control logic	2200	Controller made from JK flip-flops
TOTAL	60100	

to the preload address register which causes the address match. The critical path created by the MCB circuit is thirteen gate delays, for both the preload and the store instructions.

An approximation of the transister count for the prototype implementation in CMOS technology is listed in Table 5.1. The transister count is based upon a MCB employing 64 MCB address registers, divided into 32 sets of 2 address registers. Each address register is required to store a 32-bit effective preload address. The table also assumes 32 integer registers and 32 floating-point registers for the register number storage requirement computation. It is clear from the table that address storage requires the greatest number of hardware resources. When the MCB design is scaled upward to include more address registers, mainly the address register part of the component will scale linearly. Other hardware increases will be in the hash function and chooser section due to the increase in bits to index into the address registers.

5.2 Compiler Aspects of Memory Conflict Buffer

To take full advantage of the MCB support, a compiler must be able to intelligently reorder ambiguous store/load pairs, insert check instructions, generate correction code, and effectively

allocate registers for preloads. In this section, we discuss the compiler algorithms for exploiting the MCB support.

5.2.1 MCB instruction scheduling

To allow a selected load to move above an ambiguous store, the dependence arc between the two instructions is removed. This allows the compiler freedom to convert the load into a preload and to schedule dependent instructions of the preload above ambiguous stores. Then, a check instruction is placed immediately after the first ambiguous store the preload moved above. The code scheduler is restricted to not scheduling ambiguous stores and the check in the same cycle. Also, virtual register renaming must be performed to preserve all source operands used within the conflict correction code that are destroyed by antidependent instructions.

The basic MCB scheduling preprocessing involves the following steps:

(1) Build memory dependence graph.

(2) For each load, remove memory dependences selected by heuristics and add the source of the store of the removed dependence into the load's ambiguous store set (AS).

(3) Add one check instruction for each load with at least one memory dependence arc removed.

(4) Build complete dependence graph.

The heuristics for memory dependence removal involve picking cases in which conflicting addresses are unlikely. Three major classes of heuristics are used with success. One, the dependence of memory instructions using the same register with different define-reach is removed. The rationale behind this heuristic is that if the addresses are computed differently, then they must usually point to different addresses. Two, even though the address bases are different reg-

isters, we remove the dependence between memory instructions with different integer offsets. Three, indirect accesses are assumed to always differ from one another.

The list scheduling algorithm is modified to accommodate the check instructions. The following changes are made:

(1) When scheduling a load L, if any AS member is not scheduled yet, then confirm L as preload, else L is an ordinary load.

(2) When scheduling a check, if the corresponding load is an ordinary load, then delete the check from the instruction sequence, else schedule the check as an instruction.

(3) Keep track of the set of stores scheduled thus far.

(4) Preserve source operands of re-executed instructions.

5.2.2 Conflict correction code

The compiler provides conflict correction code for each preload instruction. The conflict correction code for a preload is invoked by its corresponding check instruction when conflicts have been detected. The correction code repairs program execution of the preload and all dependent instructions up to the point of the check. The dependent instructions of the preload may be re-executed provided there are no antidependent instructions which overwrite their source operand values. The compiler enforces this property by virtual register renaming and extending the live range of source registers until the check instruction.

Generation of the conflict correction code involves the following steps:

(1) For every preload P, mark P and all of P's flow-dependent instructions between P and its check.

(2) Copy marked instructions into a recovery block.

(3) Generate a label for the recovery block.

(4) Deposit the label into the check instruction.

(5) Tail duplicate fall through path to the conflict correction code.

(6) Extend register live range through check instruction to preserve the contents of the source register of re-executed instructions.

Tail duplication is performed because the superblock scheduling algorithm is used, and superblock prohibits any side entrances into the superblock. Tail duplication is also necessary to preserve superblock's scheduling scope.

5.2.3 Register allocation

Aggressive exploitation of the MCB support can increase the apparent register pressure. The problem is that the preload destination registers tend to have long live ranges because they are apparently used by the check instruction. However, a clever register allocation method can eliminate much of the increase in register pressure by making the following observation:

> If a preload destination register is recycled to hold the result of a nonpreload instruction, the contents of the corresponding address register remain active for the preload.

This is because only preload, store, and check instructions can affect the contents of the MCB. Therefore, while the preload destination register now holds the result of a nonpreload instruction, the corresponding address register continues to monitor conflict conditions for the preload.

Therefore, the prolonged live range for preload destination registers affects only register recycling among preloads. This observation leads to the following modified form of Chaitin's graph coloring algorithm [34].

The modified algorithm builds the interference graph in two steps. In step one, an interference graph is built among all preload destination registers. At this point, a preliminary register allocation identifies the preload registers that can be potentially spilled due to insufficient physical registers. The preloads whose destination registers are spilled are moved back to their original positions. All of the remaining preloads are now guaranteed to reside in physical registers for their whole lifetime.

In the second step, the nonpreload-destination live ranges are added into the interference graph. The interference graph is constructed again with all check instructions ignored. Note also that live ranges are extended for the source operands of the instructions which are flow dependent upon the preload instruction. The live range is extended to the conflict correction code pointed to by the corresponding check instruction. This is to prevent antidependence from forming during the physical register allocation phase. Antidependence can destroy register content which is needed for recovery. The new interference graph is merged with that produced in the first step by taking the union of all of the edges. The final register allocation is then performed by giving preload destination live ranges much higher priority than the other live ranges.

The modified register allocation algorithm is best illustrated with an example. In Figure 5.7, as far as other preloads are concerned, the preload destination virtual register $vr1$ is live until *check vr1*. The interference produced in step 1 is shown in Figure 5.8(a). According to the interference graph, $vr1$ and $vr2$ must be assigned into different physical registers, $r1$ and $r2$ in

```
preload    vr1 <-            mapping vr1 to r1
preload    vr2 <-            mapping vr2 to r2
store                        mapping vr3 to r1
last-use1      <- vr1
any-define vr3 <-
last-use2      <- vr2
check      vr1
check      vr2
```

Figure 5.7 Register assignment of an example code segment.

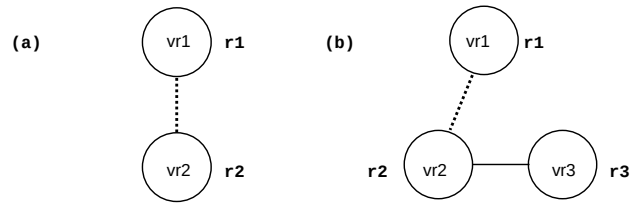


Figure 5.8 Example interference graph. (a) Interference graph for address registers only, and (b) interference graph for combined address and data registers.

this example. However, as far as the nonpreload live ranges are concerned, the live range of $vr1$ ends with *last-use1*. This is reflected in the interference graph produced in Step 2 as shown in Figure 5.8(b). Therefore, virtual register $vr3$ can reside in the same physical register as $vr1$. They are both assigned to $r1$ in our example. This is correct because the address register that corresponds to $r1$ still contains the preload address, and the conflict detection will continue until the appropriate check instruction.

5.2.4 Dependence profiling

The execution of conflict correction code incurs run-time overhead. If the overhead due to executing conflict correction code is greater than benefit of code reordering, then an overall decrease in program performance will occur. Thus, it is important to minimize the invocation of conflict correction code. This leads to the idea of *dependence profiling*, a technique to estimate

load/store address conflicts using run-time information. This information is then used by the compiler to make code reordering decisions.

The basic concept behind dependence profiling is to collect the frequency of conflicts between each ambiguous store/load pair at run time. Ideally, the address of each memory store should be compared against all subsequent load addresses. However, this would require too much storage and instrumentation time for the profiler. To solve this problem, the compiler first marks those ambiguous store/load pairs which an aggressive code scheduler would reorder to improve performance. Only the conflict status of these relevant store-load pairs is recorded during dependence profiling. In our prototype implementation, this selective dependence profiling approach successfully keeps the profiling cost under control. After dependence profiling, the compiler utilizes the conflict frequency and a conflict threshold value to make code reordering decisions.

An important benefit of dependence profiling is to minimize the negative impact of conflict correction code on the instruction cache performance. Due to dependence profiling, the invocation frequency of the correction code can be kept low. By placing all of the correction code at the end of the function, the compiler can practically eliminate the negative impact on the instruction cache performance. The performance of dependence profile for MCB has been evaluated in [35]. It will not be discussed further in this thesis.

5.2.5 Verification of transformed code

After compiler transformations, the optimizations done must be verified to ensure the program's correctness. Since formal proof of correctness is unavailable and no machine incorporating MCB has been built, the program correctness is done by inserting emulation instructions

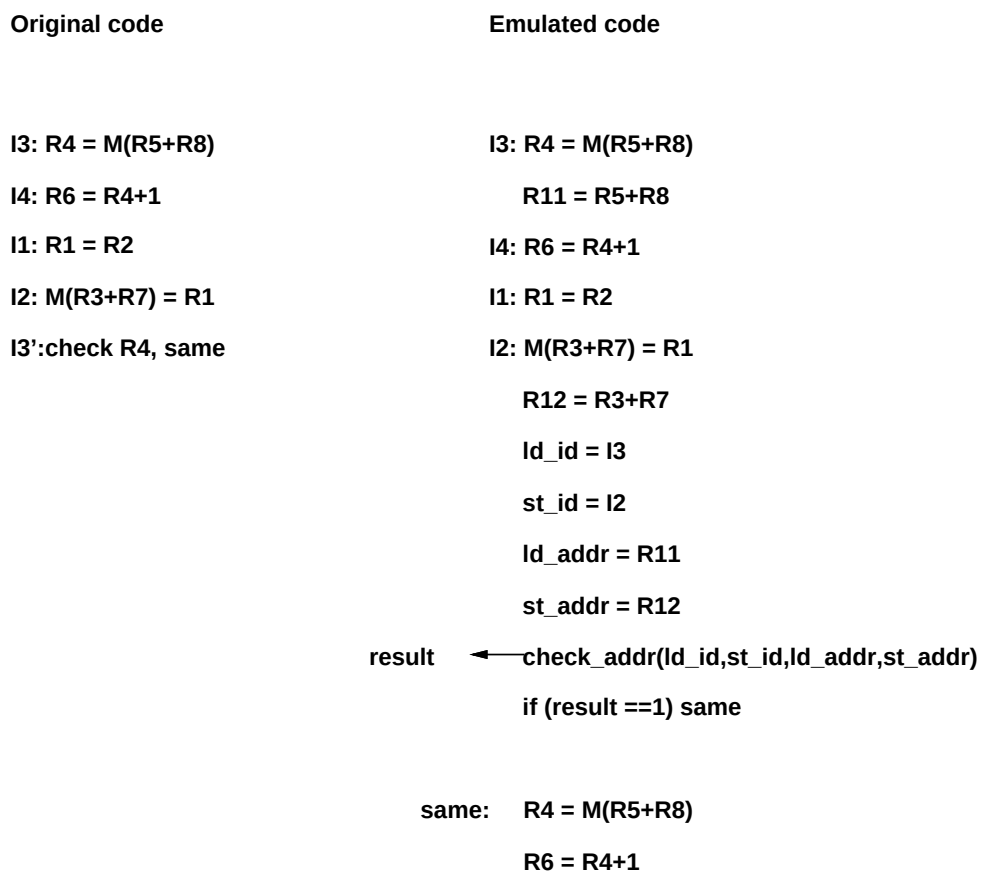


Figure 5.9 Emulation code to perform verification.

and executing the program on existing machines. Figure 5.9 shows the original code segment and the code segment with emulation instructions. Preload and store addresses are computed and preserved. The check instruction is emulated by a subroutine call which looks for address overlap. An actual conditional branch instruction is used in place of the nonexisting check. Since most architectures have no nontrapping hardware to support speculative execution, the UNIX utility signal is used to ignore all exceptions. The program output is compared against correct program output for verification.

5.3 Experimental Evaluation of Memory Conflict Buffer

Memory conflict buffer can assist many code reordering techniques, including trace scheduling [36], superblock scheduling [37], and software pipelining [38], [39]. In this section, we report our evaluation results based on superblock scheduling.

5.3.1 Evaluation methodology

The compiler algorithms described in Section 5.2 have been implemented in the IMPACT-I compiler. The performance implication of MCB is evaluated with a benchmark set that consists of nine control intensive nonnumeric programs, *cmp, compress, eqn, espresso, grep, qsort, wc, xlisp*, and *yacc*. The processors modeled in the simulation are superscalar architectures that are capable of dispatching four and eight instructions per clock cycle. The function units are pipelined and uniform for all issue rates. Latencies for all instructions are the same as in Table 4.4 except that the load and preload latencies are fixed at two. The processor includes 64-entry integer and 64-entry floating-point register banks.

5.3.2 Check instruction characteristics

Table 5.2 lists for each benchmark the total dynamic number of check instructions executed and the percentage of the total dynamic instructions that are checks. The percentage of check instructions ranges from 2.3% to 11.5% in a four-issue machine and 2.7% to 20.9% in an eight-issue machine. The number of check instructions executed can account for a large percentage of the instruction overhead due to MCB support. This amount of dynamic instruction increase requires a high issue rate architecture to absorb the cost of added instruction bandwidth. The table also lists the number of true conflicts, as would be seen by the full-scale MCB design.

Table 5.2 Check statistics.

	Issue 4			Issue 8		
Benchmark	Checks	% of Total	True Conflicts	Checks	% of Total	True Conflicts
cmp	284302	11.5%	0	576592	20.9%	0
comp	211889	4.0%	0	245846	4.4%	0
eqn	156043	4.9%	24	156979	4.9%	24
espresso	1303616	4.7%	23449	1323437	4.7%	21928
grep	108938	7.4%	0	108938	7.2%	0
qsort	5270713	9.0%	0	7875682	12.4%	0
wc	157469	9.3%	0	170697	10.0%	0
xlisp	616897	7.6%	20177	710271	8.3%	20177
yacc	412302	2.3%	7	525404	2.7%	301

Looking at Table 5.2, eqn, espresso, xlisp and yacc are the only benchmarks in which true conflicts are observed. Although some of the conflicts are due to poor judgement in dependence removal during MCB scheduling, most of these true conflicts are due to conflicting pointers and structure accesses.

5.3.3 Full-scale MCB evaluation

The speedup of the benchmarks with MCB support over a four-issue machine without MCB support is presented in Figures 5.10 and 5.11. There are three points associated with each benchmark. The connected points between the benchmarks in the middle represent the base architecture speedup without MCB support. For the remaining two points for each benchmark, the upper point present the benchmark speedup with MCB over the base. As for the lower point, MCB support is not included in a less expensive version of the same architecture. However, for compatibility reasons, the same program must execute on all versions of the architecture. The lower point shows the worst-case performance in which every check instruction executed is taken; the full penalty of conflict correction code is taken.

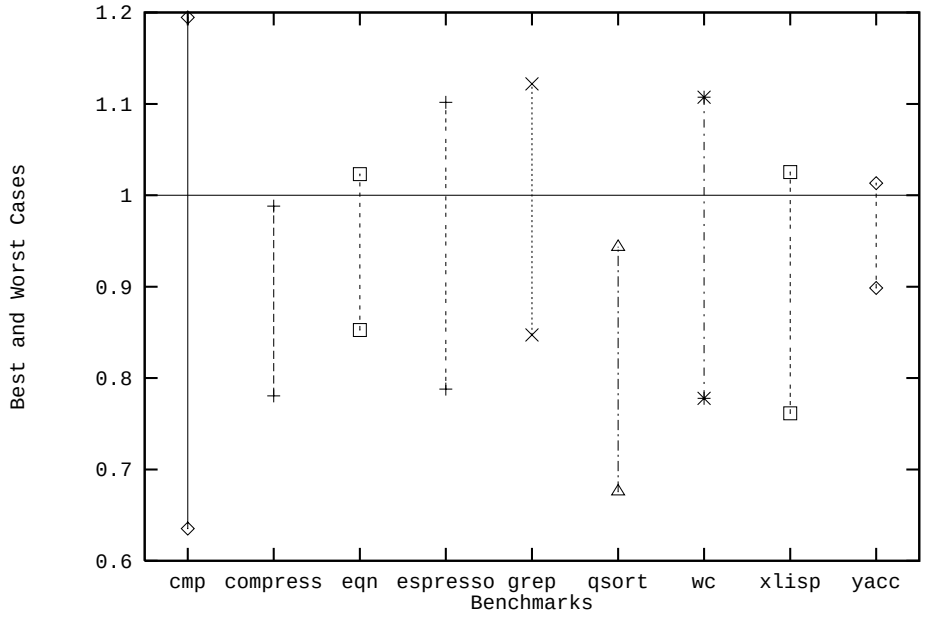


Figure 5.10 Best and worst cases, 4 issue.

The positive performance increase ranges from over 2% to 19% over a four-issue base. *Compress* and *qsort* are the only benchmarks which resulted in a performance degradation on a four-issue machine (Figure 5.10). After closer examination, the problem is different for both benchmarks. For *compress*, static branch prediction is the problem. A frequently executed superblock is exited early most of the time. The more aggressive MCB scheduling algorithm prolonged the execution time before the exiting branch by one cycle as compared to the no-MCB scheduling scheme. Therefore, the performance degradation is not a result of executing the conflict correction code, but it is actually a product of a less ideal schedule. Benchmark *Qsort*, on the other hand, takes its performance hit with an overuse of physical registers. Due to a more aggressive schedule and the preservation of the source operands, *qsort* experiences slightly more register pressure and therefore more register spills. The register spill code lengthens the schedule, thus resulting in the performance loss.

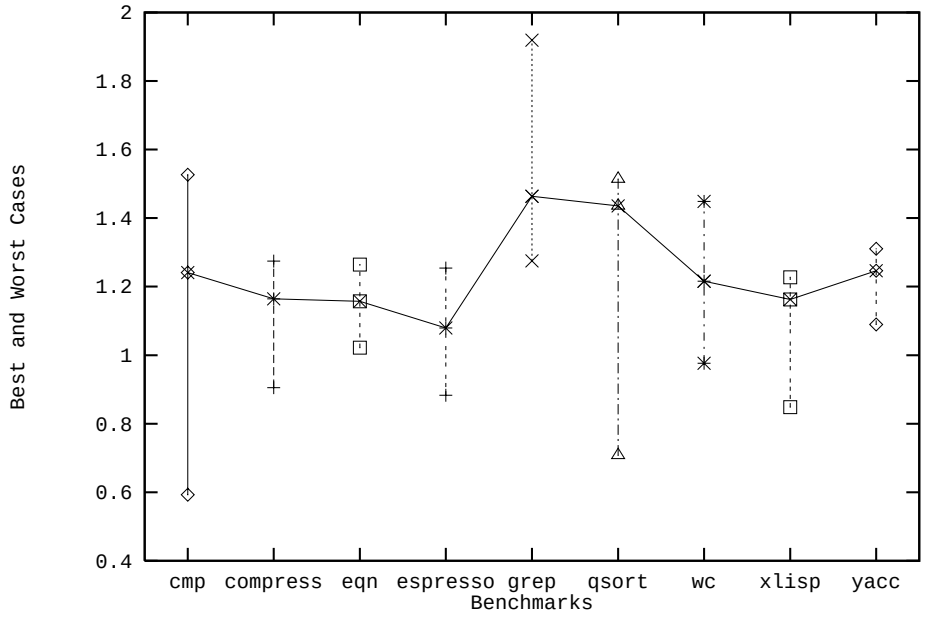


Figure 5.11 Best and worst cases, 8 issue.

Increasing the issue rate from four to eight also increases performance, as seen in Figure 5.11. Without MCB support, the speedup over a four-issue base ranges from 7% to 46%. However, additional performance gain is observed by providing MCB support, giving performance gain ranges from 24% to 91%. Eight-issue machines benefit more from MCB than a four-issue machine due to a greater need for independent instructions and more available instruction slots to hide the overhead of check instructions.

5.3.4 Hash MCB evaluation

Hash MCB design provides a less expensive alternative. However, due to its dynamic nature, the effectiveness of the hash design must be evaluated. The hashing function used in the evaluation is a permutation-based hash function. Each bit in the hash address is computed by XORing several different preload/store address bits. Mathematically, it can be represented as

Table 5.3 Percentage of checks taken due to ld/ld or ld/st conflicts, issue 4 .

Benchmark	Conflict Type	Number of Hash Entries							
		8	16	32	64	128	256	512	1024
cmp	ld/ld	3.30%	1.64%	0.82%	0.41%	0.21%	0.11%	0.06%	0.04%
	ld/st	3.77%	1.82%	0.91%	0.46%	0.22%	0.11%	0.05%	0.03%
compress	ld/ld	9.22%	7.98%	0.88%	0.43%	0.22%	0.11%	0.05%	0.03%
	ld/st	11.8%	4.89%	2.73%	1.42%	0.71%	0.35%	0.17%	0.11%
eqn	ld/ld	0.43%	0.19%	0.10%	0.05%	0.03%	0.03%	0.02%	0.02%
	ld/st	10.1%	4.21%	2.41%	1.11%	0.54%	0.28%	0.14%	0.09%
espresso	ld/ld	21.8%	15.8%	12.1%	10.7%	10.0%	9.67%	9.56%	9.50%
	ld/st	9.48%	6.16%	3.85%	3.10%	2.20%	2.09%	2.03%	1.97%
grep	ld/ld	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	ld/st	3.62%	2.08%	1.04%	0.70%	0.46%	0.33%	0.09%	0.02%
qsort	ld/ld	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	ld/st	15.0%	12.2%	3.00%	1.58%	0.85%	0.30%	0.12%	0.01%
wc	ld/ld	0.31%	0.16%	0.08%	0.04%	0.02%	0.01%	0.00%	0.00%
	ld/st	6.93%	3.43%	1.71%	0.85%	0.42%	0.20%	0.10%	0.04%
xlisp	ld/ld	7.95%	4.35%	1.87%	0.90%	0.48%	0.34%	0.29%	0.21%
	ld/st	14.2%	9.60%	6.64%	5.61%	4.90%	4.32%	4.18%	3.96%
yacc	ld/ld	8.58%	6.85%	2.82%	1.42%	1.21%	1.08%	1.00%	0.99%
	ld/st	8.81%	5.37%	2.77%	0.83%	0.39%	0.26%	0.17%	0.13%

matrix multiplication where *hash_address* = A * *address*. The matrix **A** used in the simulation is

$$\begin{matrix} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix}$$

The hash address generated is scaled down for the appropriate number of hash locations. Also, the least significant bits of the memory address are truncated to support byte, half-word, and full word accesses.

Table 5.4 Percentage of checks taken due to ld/ld or ld/st conflicts, issue 8.

Benchmark	Conflict Type	Number of Hash Entries							
		8	16	32	64	128	256	512	1024
cmp	ld/ld	4.59%	2.20%	1.09%	0.55%	0.26%	0.12%	0.05%	0.03%
	ld/st	5.12%	2.40%	1.21%	0.61%	0.29%	0.13%	0.07%	0.03%
compress	ld/ld	8.10%	6.94%	0.77%	0.39%	0.20%	0.10%	0.05%	0.03%
	ld/st	11.9%	3.99%	2.21%	1.15%	0.57%	0.28%	0.14%	0.10%
eqn	ld/ld	0.43%	0.18%	0.10%	0.05%	0.03%	0.02%	0.02%	0.01%
	ld/st	10.1%	4.26%	2.47%	1.12%	0.55%	0.29%	0.15%	0.10%
espresso	ld/ld	20.9%	15.5%	11.7%	10.6%	9.66%	9.29%	9.17%	9.18%
	ld/st	8.99%	5.23%	2.87%	2.22%	1.93%	1.80%	1.74%	1.68%
grep	ld/ld	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	ld/st	5.62%	3.13%	1.60%	0.98%	0.59%	0.38%	0.12%	0.03%
qsort	ld/ld	1.29%	1.15%	1.07%	1.07%	0.00%	0.00%	0.00%	0.00%
	ld/st	16.5%	10.9%	3.76%	2.36%	1.94%	1.31%	1.13%	0.91%
wc	ld/ld	0.29%	0.15%	0.08%	0.03%	0.01%	0.00%	0.00%	0.00%
	ld/st	7.87%	3.91%	1.94%	0.97%	0.48%	0.24%	0.12%	0.05%
xlisp	ld/ld	11.3%	4.58%	1.56%	0.98%	0.63%	0.39%	0.35%	0.29%
	ld/st	13.8%	8.74%	6.64%	5.22%	4.10%	3.71%	3.58%	3.25%
yacc	ld/ld	16.5%	12.1%	5.00%	2.22%	1.51%	1.39%	1.17%	1.16%
	ld/st	7.92%	4.60%	2.62%	0.94%	0.44%	0.40%	0.26%	0.22%

The hash MCB performance is dominated by the number of false conflicts. All true conflicts are detected. However, the additional false conflicts determine how well hash MCB designs perform. To give an intuitive feeling, Tables 5.3 and 5.4 shows the percentage of the check instructions that are taken due to false conflicts. The false conflicts are further divided into ld/ld and ld/st conflicts. False conflicts level off approximately around 32 to 64 hash table entries. More hash table entries do not further reduce false conflicts significantly.

The performance of hash MCB is shown in Figure 5.12 to 5.29. In addition, a performance trade-off is made on whether the extra preload instruction is actually necessary or not. By eliminating the preload instruction, all load instructions will enter the MCB hardware. A potential problem is that the false ld/ld conflict may increase, therefore degrading performance. Two curves are presented for each benchmark, partial preload and all preload. The partial

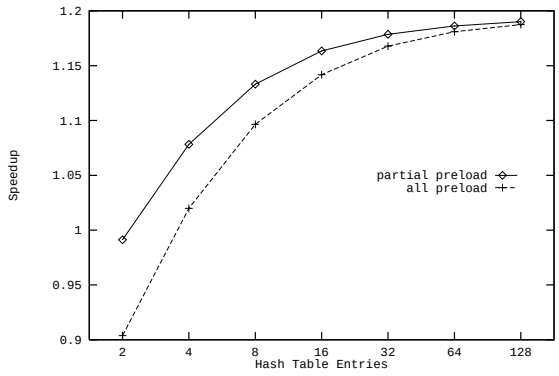


Figure 5.12 Hash MCB, cmp, 4 issue.

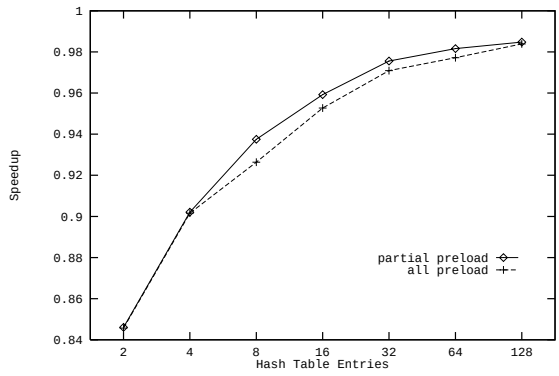


Figure 5.13 Hash MCB, compress, 4 issue

preload curve shows the MCB performance with the addition of a preload instruction. In contrast, the all preload curve allows all loads to enter the MCB hardware.

As expected, the performance curve reaches the asymptotic point around 32 to 64 MCB entries. The asymptotic hash MCB performance is comparable to the full-scale MCB. Even without the additional preload instructions, the hash MCB performance tracks the partial preload curve closely. The only problem is *espresso*. Comparing Figures 5.10 and 5.11, *espresso* achieves 10% and 25% performance improvement over the base, respectively. However, in the hash MCB design, the asymptotic performance levels off at 3% and 18% for four-issue and eight-issue. This big drop is due to ld/ld conflicts observed in Tables 5.3 and 5.4. *Espresso* uses right shifts in address computation. Therefore for several consecutive accesses, the preload addresses are the same. This causes the same hash address to be generated and causes false ld/ld conflicts. One way to avoid this is to recognize such address computation as dangerous and disallow promotion to preload in such an instance.

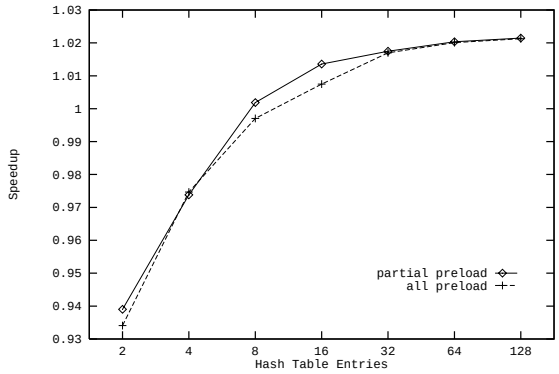


Figure 5.14 Hash MCB, eqn, 4 issue

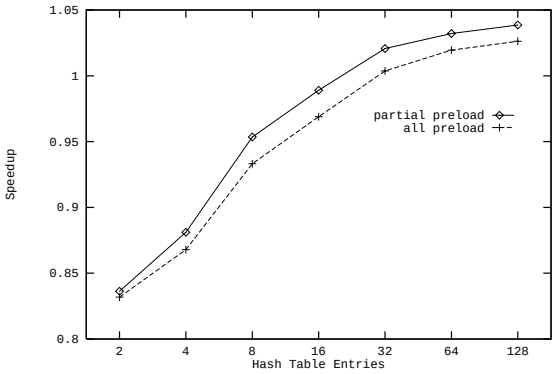


Figure 5.15 Hash MCB, espresso, 4 issue

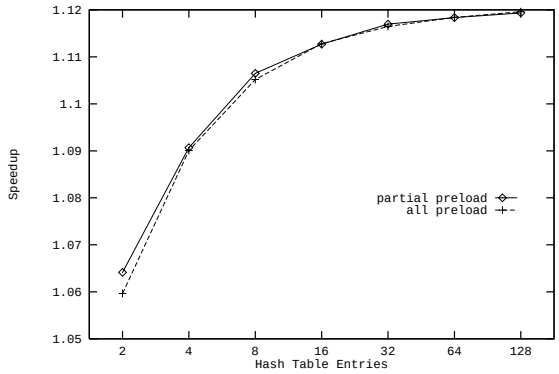


Figure 5.16 Hash MCB, grep, 4 issue

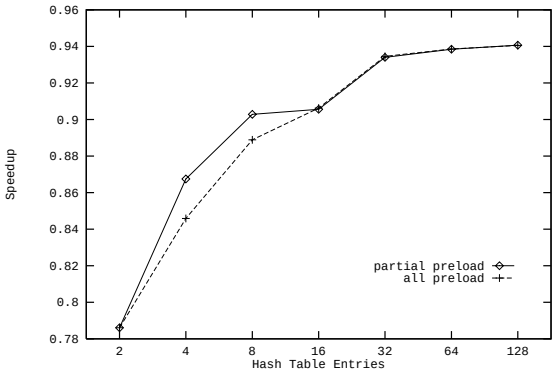


Figure 5.17 Hash MCB, qsort, 4 issue

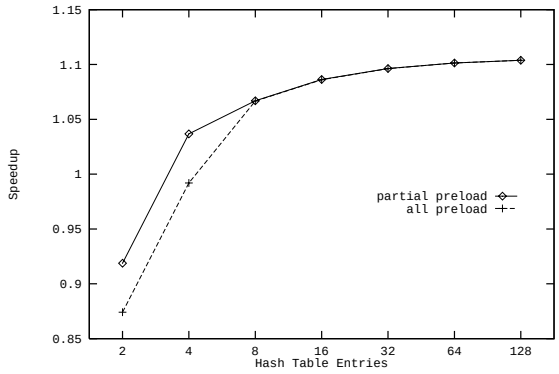


Figure 5.18 Hash MCB, wc, 4 issue

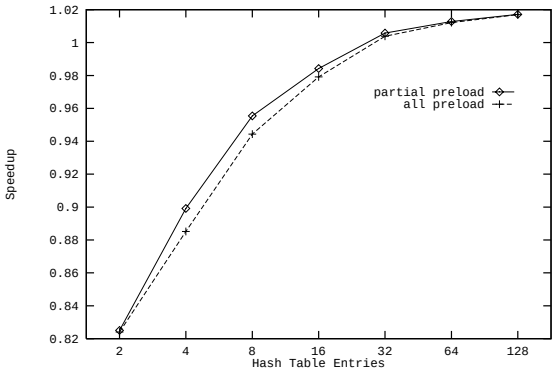


Figure 5.19 Hash MCB, xlisp, 4 issue

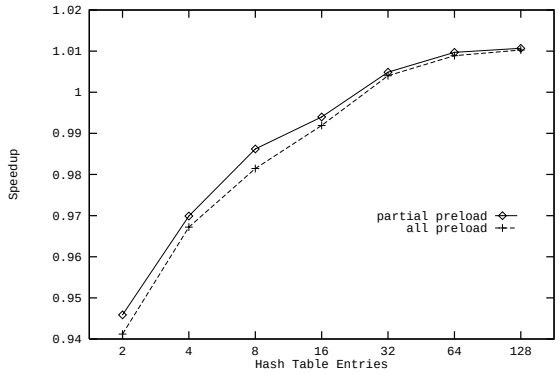


Figure 5.20 Hash MCB, yacc, 4 issue

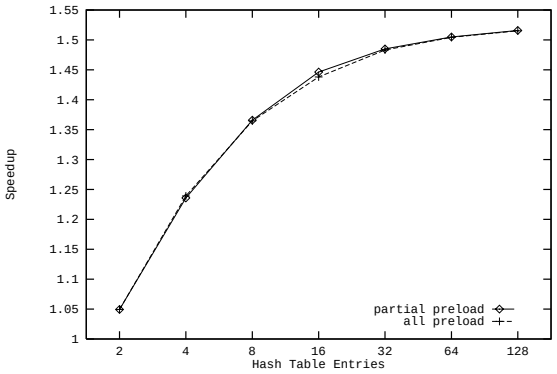


Figure 5.21 Hash MCB, cmp, 8 issue

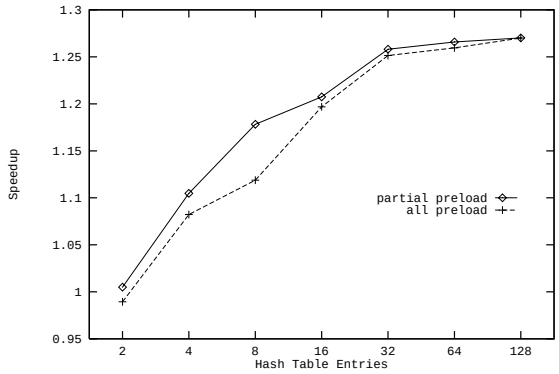


Figure 5.22 Hash MCB, compress, 8 issue

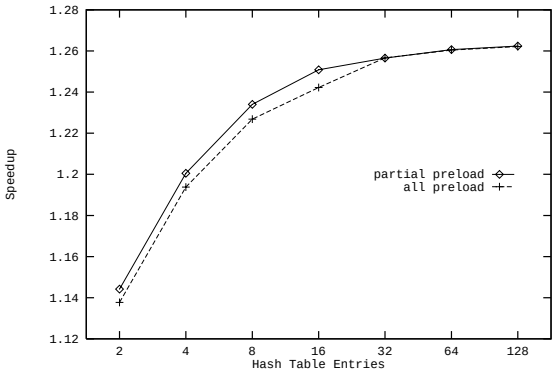


Figure 5.23 Hash MCB, eqn, 8 issue

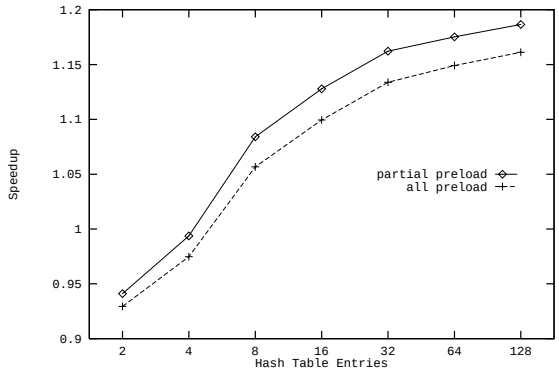


Figure 5.24 Hash MCB, espresso, 8 issue

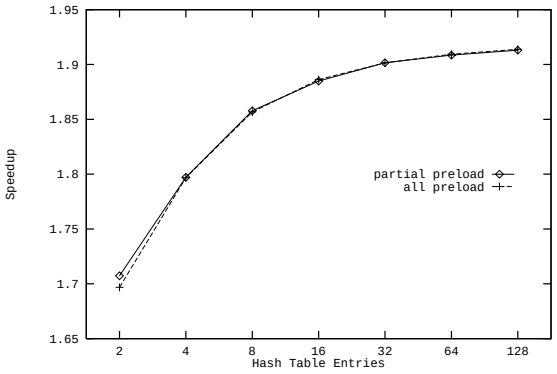


Figure 5.25 Hash MCB, grep, 8 issue

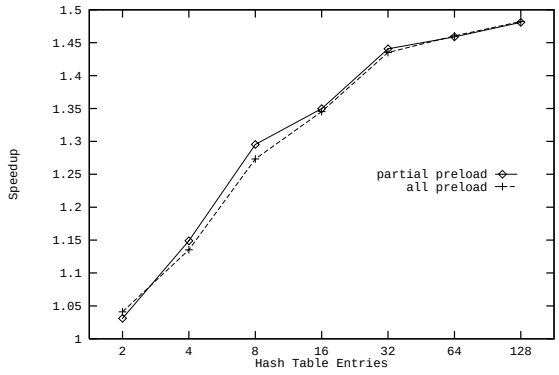


Figure 5.26 Hash MCB, qsort, 8 issue

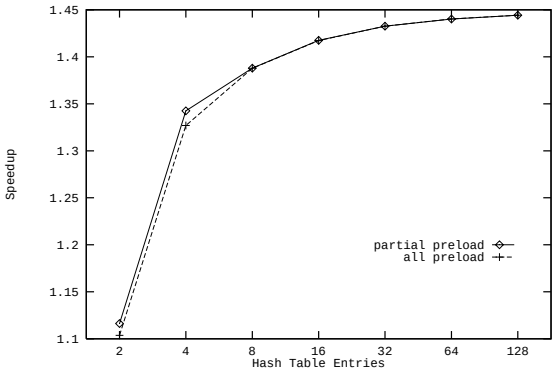


Figure 5.27 Hash MCB, wc, 8 issue

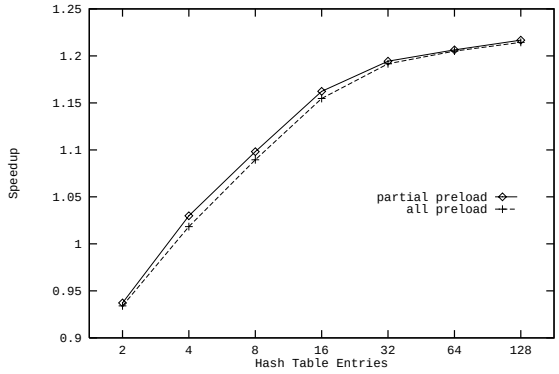


Figure 5.28 Hash MCB, xlisp, 8 issue.

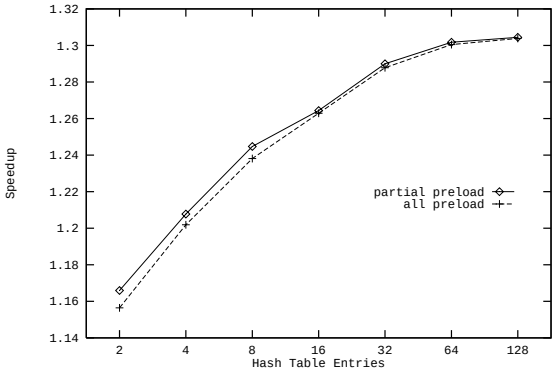


Figure 5.29 Hash MCB, yacc, 8 issue.

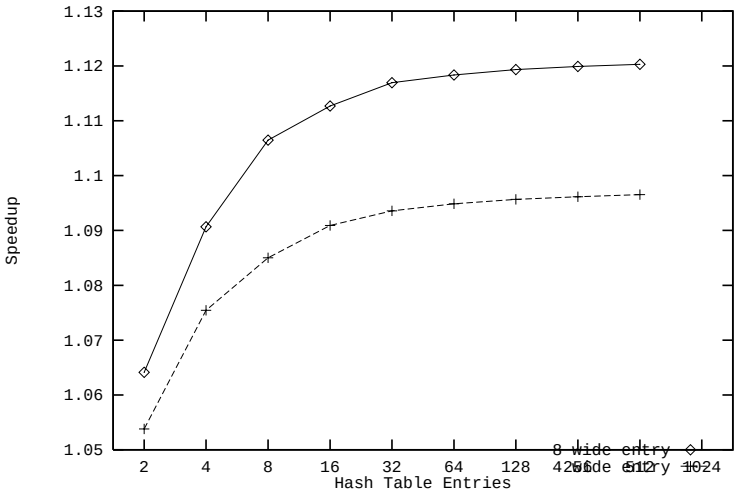


Figure 5.30 Hash entry trade-off for *grep*, 4 issue.

5.3.5 MCB entry width evaluation

When the MCB entry width is increased to accommodate different access sizes, the storage area of MCB hardware increases proportionally. Byte and word accesses are the two most common access sizes. We wish to evaluate whether an MCB entry can treat half-word accesses the same as a word access, thus reducing each MCB entry by a factor of two. Figures 5.30 shows *grep* on a four-issue processor and Figure 5.31 shows *compress* on an eight-issue processor. The top curve shows the MCB performance supporting all access sizes. The lower curve shows the MCB performance treating half-word accesses the same as word accesses. As can be seen, the lower curve suffers a large dip in performance as the number of MCB entries increases. The asymptotic performance does not track well. The difference in performance, however, is not due to half-word accesses but to sequential byte accesses. A half-word access consists of two sequential byte accesses. When half-word accesses are ignored within MCB entries, information regarding sequential byte accesses is lost. A byte access can replace another sequential byte access within an MCB entry due to the loss of information. False ld/ld conflicts increase, and

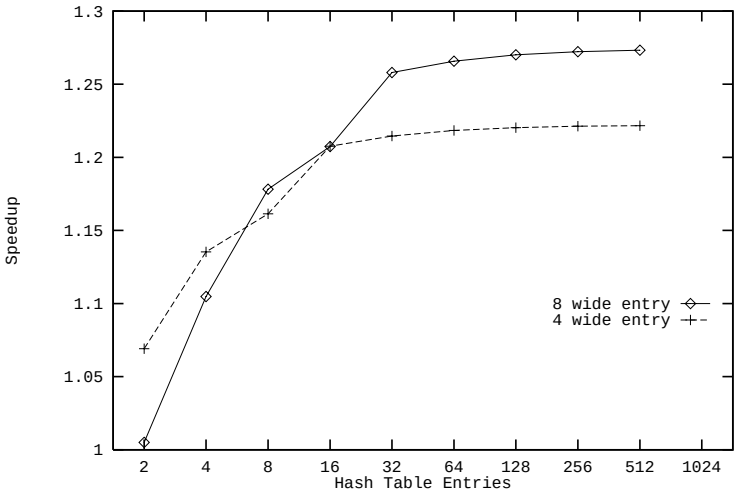


Figure 5.31 Hash entry trade-off for *compress*, 8 issue.

performance decreases. These two benchmarks serve as an example of what would happen when an MCB entry width is reduced. Therefore, full MCB entry width should be supported.

5.4 Summary of Memory Conflict Buffer

This chapter proposes an architectural feature, the memory conflict buffer (MCB) to increase the opportunities of code reordering. This is achieved by removing the dependence between a store/load pair when they seldom access the same location; thus a memory load and its dependent instructions can be moved above any number of memory stores regardless of the dependence relation between the load and the stores. The MCB hardware supports such code reordering by detecting the situation in which the ambiguous reference pair access the same location and subsequently invoking a correction code sequence supplied by the compiler to restore the correctness of the program execution. The combination of these supports enable the compiler to expose instruction-level parallelism previously restricted by the store/load dependences.

CHAPTER 6

PRELOAD BUFFER

Cost versus performance tradeoffs have resulted in system designs with fast processors, and diverse speeds of memory subsystems. The disparity between the speeds of the memory subsystems creates longer communication delays which if not carefully managed, can eliminate most of the benefits gained from a fast processor. Recently, several hardware and software solutions to minimize the effect of this communication delay, or memory latency, have been proposed. This chapter focuses on method of data preload to reduce this latency. First, a complementary method, cache prefetching, is discussed. Then, the preload buffer is introduced to compensate for the deficiency of cache prefetching.

6.1 Cache Prefetching and Memory Latency

Early cache loads with the assistance of an optimizing compiler is an effective means of reducing the penalty of long memory access time beyond the primary cache. Among those, cache prefetching is the most prominent one in recent literatures. However, cache prefetching can cause cache pollution and its benefit can be unpredictable. This section describes cache prefetch, its variations, and its problems. Through the discussion, an attempt is made at discovering what the preload support should encompass to overcome these problems.

6.1.1 Cache pollution

The main objective of data prefetching is to decrease the overall execution time of a given application. Cache prefetching is typically performed for scientific applications, where the performance of caches is often inadequate. Many scientific applications sweep through several arrays that are larger than the current cache sizes. In this case, expecting a particular element to survive replacement within the first level cache for repeated use is unlikely. Therefore, the compiler attempts to bring the data to be used in the near future into the cache through prefetching. The idea is to have the data available in the cache when the actual memory access occurs. This, however, increases the requirements of the data cache. Now, the cache is expected to hold not only the current working set, but also the future working set simultaneously. The working set size that the cache is required to hold depends on the prefetch strategy and the memory latency. If the working set requirement is larger than what the cache size can handle, additional cache pollution will occur, thus degrading the cache performance further. In addition, as cache pollution increases, off-chip memory traffic also increases. The cache behavior, however, cannot be easily determined at compile time, and thus the benefit of cache prefetching is unpredictable.

6.1.2 Prefetch into the prefetch buffer

The problem of cache pollution due to prefetching into the cache can be solved by prefetching the data into a separate prefetch buffer, which is organized as a fully associative FIFO queue [22]. The prefetch buffer can be thought of as a second cache. The data cache holds the current working set, and the prefetch buffer holds the possible future working set. The processor

therefore operates on two separate caches. When the data in the prefetch buffer are referenced, the associated cache line for the data is transferred from the buffer into the data cache.

The existence of a prefetch buffer creates two problems: the associativity of the prefetch buffer and the coherence between the data cache and the prefetch buffer. Implemented as a cache with FIFO replacement policy, the prefetch buffer size is proportional to the memory latency of the system. Its size can become unreasonably large for full set associativity as the memory latency increases. Reducing the associativity, on the other hand, increases the probability of conflict within the prefetch buffer and causes additional unnecessary memory traffic.

Maintaining coherence between the data cache and the prefetch buffer further complicates the memory system design. Extra communication channels must exist between the cache and the prefetch buffer in order for the cache to inform the prefetch buffer of any dirty data and to transfer data from the buffer to the cache. This extra channel complicates cache controller design.

6.1.3 Prefetch into registers

The concept of increasing the distance between a load instruction and the use of the load destination register is called register preloading. Since architectural registers are managed by the compiler, the registers exhibit none of the cache pollution effects. Register preloading has proved to be tolerant of the memory latency, but it achieves the objective at a large increase in register usage [40]. This increase in register usage requires either nonconventional register file design or an increase in the size of the instruction. Also, additional problems are raised with

memory dependences and conditionals within the loop body since the value preloaded into the register must be binding.

6.1.4 Prefetch for superscalar processors

To sustain its performance, a superscalar processor must be able to issue multiple memory accesses per cycle [28]. This increase in the number of memory accesses further complicates the problem of prefetching either into the cache or into a prefetch buffer. In a superscalar processor, a prefetch instruction must bypass more instructions to compensate for a given number of cycles. Since more than one memory access can be executed per cycle, increasing the number of instructions bypassed leads to more memory accesses executed between a given prefetch instruction and its associated memory load instruction. Under these conditions, further cache pollution may result. This also holds true for a lower associativity prefetch buffer, because the probability of a replacement within the prefetch buffer increases proportionally. Although a fully associative FIFO prefetch buffer can alleviate the problem, the number of entries and the size of the fully associative comparator can increase dramatically as the number of simultaneously executable memory loads increases.

6.1.5 Prefetching architecture requirements

A new architecture for prefetching has to solve the following problems:

(1) Prefetching the future working set must not replace the present working set from the cache.

(2) In maintaining coherence, the architectural support for prefetching must be disjoint from the data cache to avoid increasing the bandwidth and complexity of the data cache.

(3) Associativity should be disallowed to avoid comparison overhead and reduce cycle time.

(4) Prefetched data must not be replaced or discarded unless it is deemed unnecessary.

(5) The architecture must be expandable as the number of simultaneously executable memory loads increases per processor cycle.

This chapter presents the preload buffer based on the above criteria. This support is shown to be effective in reducing the miss ratio, the memory latency, and the memory traffic for a set of scientific benchmarks. The next section further explains the cache prefetching problem with simulation results. In Section 6.3, the new preload architecture and compiler support are described in detail. An algorithm for prefetching and preloading loops is presented in Section 6.4. Section 6.5 presents the simulation result of the new prefetch architecture. Concluding remarks are given in Section 6.6.

6.2 A Brief Empirical Study of Cache Prefetching

This section focuses on the inherent problems associated with cache prefetching. An infinite-sized prefetch buffer is used in the simulation as a basis for the lower bound on the miss ratio. Cache prefetching will be compared against our proposed preloading scheme using more realistic parameters in Section 6.5.

All of the results reported are obtained using the execution-driven system simulation based on actual compiled code. Within the context of this chapter, an issue-4 processor model has been chosen to represent the base architecture. No restriction is placed on function unit usage. As with the issue rate, the instruction latency also affects where the prefetch instruction is inserted for a given memory latency. The instruction latencies shown in Table 4.4 are used for

Table 6.1 Miss ratio of benchmarks.

Benchmark	*Miss Ratio*	
	16K Cache	*256K Cache*
doduc	6.65%	0.15%
fpppp	3.15%	0.02%
matrix300	23.13%	13.66%
nasa7	31.82%	15.36%
tomcatv	25.59%	22.05%

our simulation (loads are fixed at two processor cycles). The assumed cache miss latency is 30 processor cycles. The simulation results reported in this section are based on single-level direct-mapped write-through caches with a block size of 32 bytes. These caches are nonblocking with no limit on outstanding requests.

Benchmarks with a significant level of cache misses should be used to evaluate prefetch schemes, since cache prefetch is needed only when the miss ratio is high. The miss ratios for five benchmarks are shown in Table 6.1. Two of the benchmarks, *doduc* and fpppp, exhibit relatively low miss ratios. By increasing the cache size, the misses almost completely disappear. Therefore, these benchmarks will not be discussed further. High miss ratios, however, are observed for *matrix300*, *nasa7*, and *tomcatv*, even with a 256K cache. Since increasing the cache size does not sufficiently reduce the miss ratios, cache prefetching is considered as an alternative approach to improving the memory access behavior. Further examination of the seven independent kernels from *nasa7* finds that five of them, *fft2d*, *cholsky*, *btrix*, *gmtry*, and *vpenta*, have high miss ratios. These five *nasa7* kernels are studied separately along with *matrix300* and *tomcatv*.

Figure 6.1 compares the miss ratios with and without prefetch support for three cache sizes. The results of the fully associative prefetch buffer are shown by the black bar. The data, however, are not transferred to the cache from the prefetch buffer upon use. The black bar

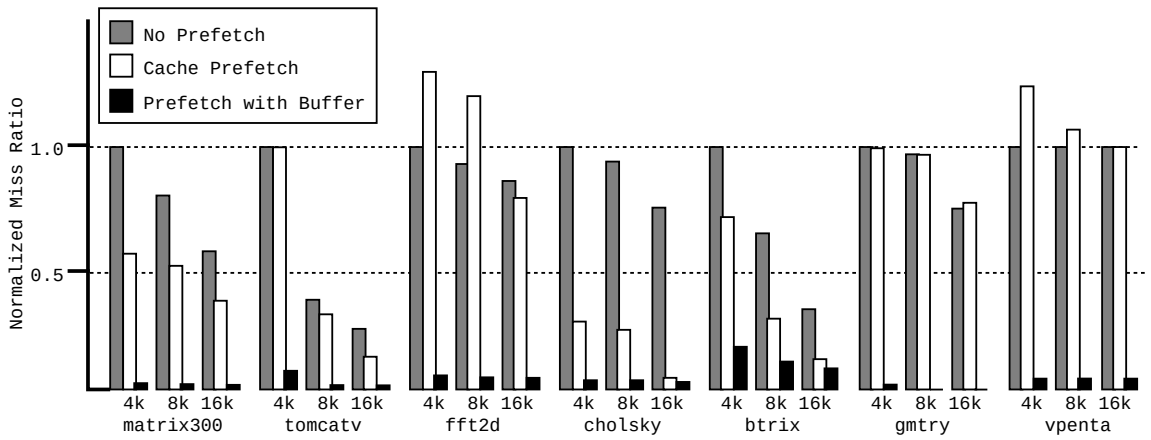


Figure 6.1 Comparison of normalized miss ratios.

closely represents the ideal prefetch case and serves as a good basis for improvement over the other two results. The black bar has nonzero results due to the start-up cost of prefetching and the fact that non-inner-loop accesses are not prefetched. For each benchmark, the miss ratio is normalized based on results for a 4K cache with no prefetching.

The effectiveness of cache prefetching is mixed with respect to the miss ratio. Prefetching into the cache substantially reduces the miss ratio for all of the cache sizes shown for *matrix300*, *cholsky*, and *btrix*. Most of the prefetched data for these three benchmarks are used before replacement. In the case of the other four benchmarks, prefetching into the cache offers either little improvement or actually degrades the cache performance. With a separate prefetch buffer to eliminate conflicts between current and future working sets, prefetching can potentially reduce the miss ratio significantly as shown by the black bar for all benchmarks. This graph has clearly shown that the problems of block replacement and pollution are quite significant when cache prefetching is performed. The pollution effect causes the behavior of cache prefetching to be unpredictable. This is an undesirable effect since the performance degradation can be large.

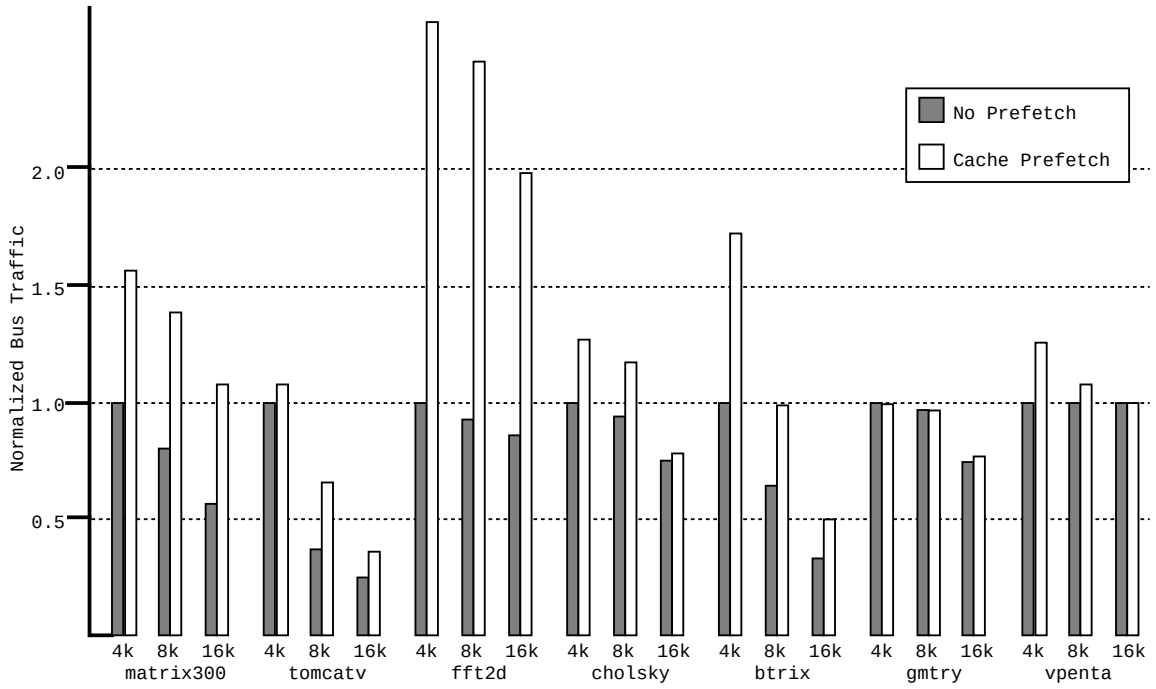


Figure 6.2 Comparison of normalized bus traffic.

Miss ratio alone does not give a complete picture of the effects of cache prefetching. Bus traffic must also be evaluated with respect to system performance. Figure 6.2 plots the normalized bus traffic for the no prefetch and prefetch cases.[1]

It is not surprising that the benchmarks with increased miss ratios due to prefetching exhibit a dramatic increase in bus traffic. We are also interested in the benchmarks with a decrease in miss ratio, specifically *matrix300*, *cholsky*, and *btrix*. In all cases, the bus traffic ratio is higher than without prefetching. This increase in bus traffic occurs when data elements in the current working set are replaced by prefetched data elements and later reloaded from memory. Depending on the amount of additional bus traffic, the overall benefit of prefetching can degrade proportionally.

[1] This model assumes that prefetched data are not cached to ensure that there are no conflicts between the current and future working sets. As a result, the miss ratio is low but the penalty for bus traffic is high since every prefetch is required to go off-chip. The bus traffic associated with an ideal case prefetch buffer is therefore omitted from the chapter.

To summarize, prefetching into the cache increases cache pollution which in turn creates additional bus traffic. The overall performance gain can be positive or negative, which leads to unpredictable memory system behavior. The new prefetch architecture support must deal with these issues. The effectiveness of the new architecture will be compared with cache prefetching in Section 6.5 using a more detailed system simulation.

6.3 The Preload Buffer

The preload buffer is designed to solve the problems discussed previously. It has the following characteristics:

- All preload data are stored within the preload buffer. The data in the preload buffer are not transferred to the cache when used.
- The preload buffer is a separate module from the data cache. No communication between the data cache and the preload buffer is necessary.
- The preload buffer is employed to complement cache prefetch.
- No comparison of address tags is necessary to retrieve a preload buffer entry. All entries are directly accessible using an index.
- All preload data are mapped to some preload entry and will not be replaced by another preload unless they are used or become unnecessary.
- The bus traffic for each loop iteration is predictable.
- The preload buffer is expandable with increasing instruction issue rate. Multiple accesses to the preload buffer are always possible without bank conflicts.

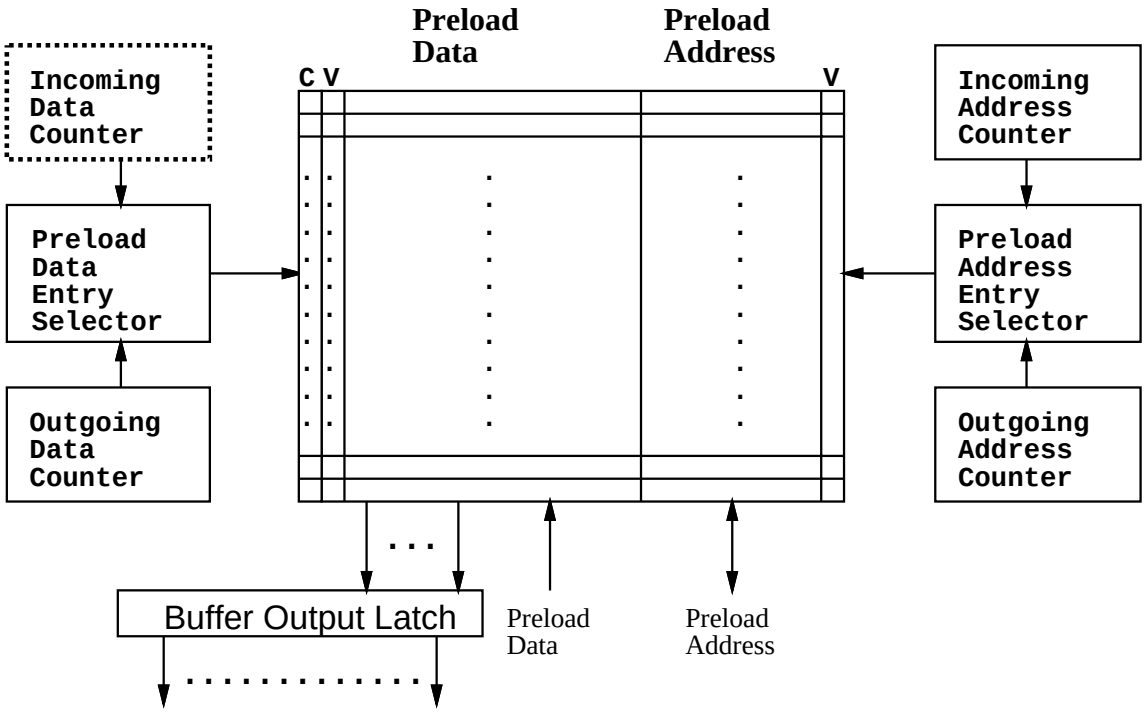


Figure 6.3 Hardware support for the preload buffer.

6.3.1 Hardware organization

The hardware support for preload buffer is shown in Figure 6.3. It consists of a bank of data registers, a bank of memory address registers, and several data and address entry counters. The data registers are used to hold preload data, and the memory address registers are used to hold the effective preload addresses for data accesses. Each counter serves as an index into the preload buffer. The incoming counters are used to place an item within the preload buffer. The outgoing counters are used to obtain an item from the preload buffer. The (V) bits indicate when the preload address and preload data entries are valid. The (C) bit indicates the possibility of a stale datum within the preload buffer. The number of entries within the preload buffer is an architectural parameter that must be made known to the compiler to prevent erroneous program execution.

Several new instructions are introduced into the instruction set to utilize the hardware. They are *init*, *preload*, *loadb*, *prestore*, *storeb*, and *skip*. Except for the init instruction, input operands are specified for the other instructions. They have the following formats:

preload *offset(base-address)*

loadb *dest, offset(base-address)*

prestore *offset(base-address)*

storeb *offset(base-address)*

skip *value*

The loadb instruction exercises the preload buffer instead of the cache. Otherwise, the loadb instruction has the same semantics as a regular memory load instruction. The effective address is calculated by adding the offset to the base address. The operand for skip, *Value*, is an integer specifying the increment for the outgoing data counter. We proceed to the discussion of the preload buffer operation for a loop without conditionals and without memory hazards using the *init*, *preload*, and *loadb* instructions. Later in this section, the *prestore*, *storeb* and *skip* instructions will be discussed as well as how they are used to deal with memory hazards and conditionals.

6.3.2 Basic concept

The preload buffer serves the memory loads within a single loop iteration as sequential accesses. Each load operation should appear to the preload buffer in its instruction fetch order. It is therefore possible to map each memory load into some sequential entry within the preload buffer. Preloads to the preload buffer can be viewed as "pushes onto the queue" and loads from the preload buffer as "pops from the queue."

Before executing any preload instructions, the *init* instruction is executed to reset the counters. All entries of preload data and preload addresses are also invalidated. The preload buffer is now ready for operation.

The *preload* instruction places the calculated effective address within the entry pointed to by the incoming address counter. The preload address field of the entry is marked as valid. For superscalar processors which can execute multiple preloads simultaneously, the effective addresses are inserted sequentially into the address registers in their instruction fetch order. The incoming address counter is then incremented by the appropriate offset equal to the number of addresses inserted. The address counter wraps around as it is incremented past the last preload address entry. Note that the preload buffer is a circular queue, and all addresses and data are placed accordingly. The preload instruction does not affect any other counters.

The preload buffer arbitrates for the system bus with the write buffer and data cache through a unique time stamp. All memory writes and reads should appear in order to ensure correctness. The current preload to arbitrate for the system bus is a valid address entry pointed to by the outgoing address counter. When the preload buffer obtains the system bus, the transmitted packet contains the preload address and the preload data entry location. The preload data entry location is used to latch the returning data into the correct preload buffer entry. Although this method allows out-of-order return of preload data, it incurs additional bus traffic from the sending and receiving of the preload data entry location. An alternative solution is to provide an incoming data counter and permit only the memory system to return in-order preload data. We assume the existence of an incoming data counter for our simulation results. The preload address field of an entry is marked invalid when the request is sent.

The preload data field of an entry is set to valid when the requested data are received. Due to the lag time in the processing of the preloads, the incoming address counter will always lead the outgoing address counter with the incoming data counter trailing. All subsequent memory loads to the preload buffer obtain their data from the entry pointed to by the outgoing data counter. If the preload data entry is invalid and the associated memory register is valid, the memory request has not returned and the access must wait for the request to return. When the data are read, the entry is marked invalid as a side effect.

Due to memory coherence or context switching, data in the preload buffer may be incorrect or even unavailable. This is indicated by the coherence bit (C) of the preload buffer. If the coherence bit of the preload data entry is set, the load request is routed to the data cache. The memory load instruction provides the effective address as usual. Details of the coherence bit are explained in Sections 6.3.3 and 6.3.6. The outgoing data counter increments and wraps around in a manner similar to all other counters of the preload buffer.

Since preload instructions are inserted for all loop iterations, the last few iterations of the loop may contain useless preloads. Upon exiting the loop, the *init* instruction is executed again. The main purpose at this time is to invalidate all of the preload address registers. This prevents the preload buffer from sending residual preload requests.

Due to system bandwidth and lag time, different spacings between each counter are anticipated. To clarify the relationship between the four pointers, an ordering between the address and data counters is shown in Figure 6.4. The operation of the preload buffer is shown with a simple example given in Figure 6.5. The states of the preload buffer are shown from a to h for the code segment given.

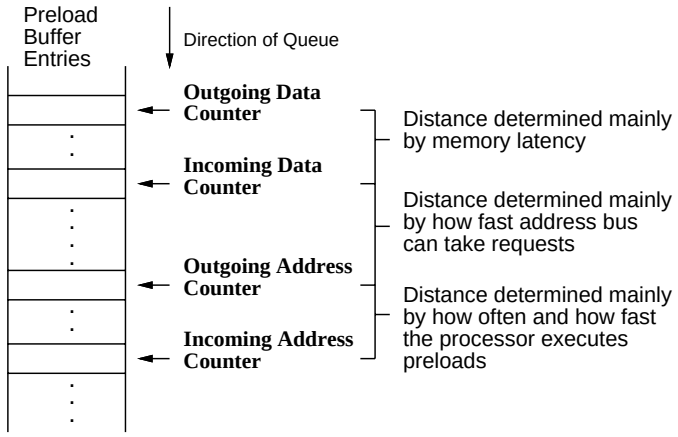


Figure 6.4 An ordering for the address and data counters.

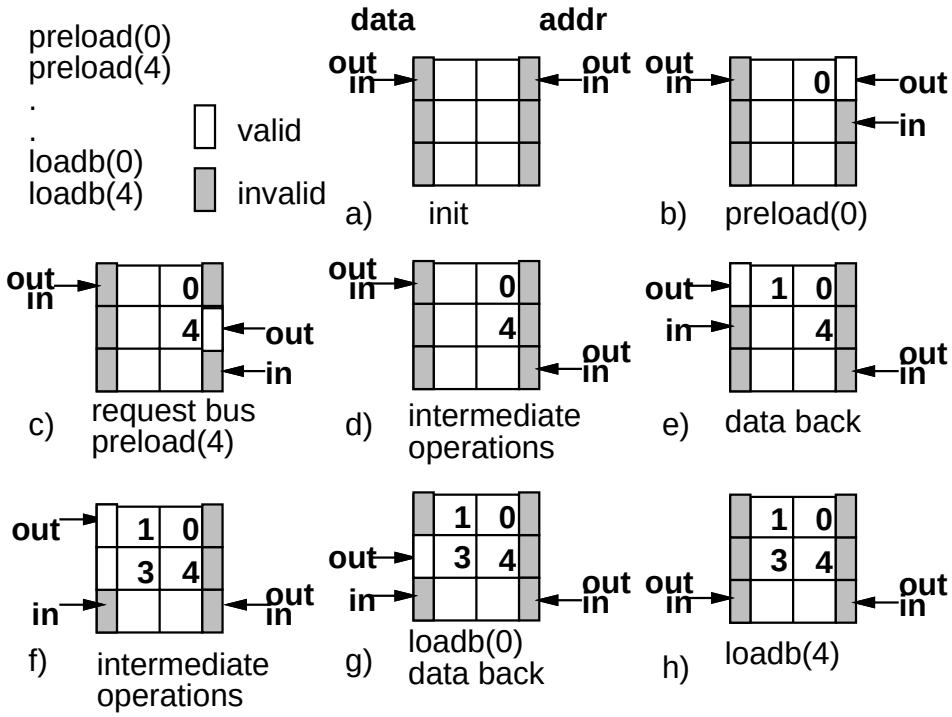


Figure 6.5 An example operation of the preload buffer.

Instructions	
preload	0(60)
preload	0(20)
store	0(60),r5
preload	0(60)
store	0(20),r3
loadb	r8,0(60)
loadb	r7,0(20)
preload	0(20)

Figure 6.6 Example coherence problem.

6.3.3 Memory dependences

Memory dependences pose a difficult problem for the preload buffer. For simplicity and speed, the preload buffer does not maintain coherence with the data cache. All values preloaded are binding and may not necessarily reflect the current memory state. If a memory location is preloaded and later changed by a store, the value obtained by a later load is incorrect. This problem is illustrated by the left column of Figure 6.6. Since addresses 20 and 60 were preloaded and later updated by a stores, the values in the preload buffer are stale. Thus, when the loadb instructions for these preloads are executed, they will load the stale data.

To ensure correct program execution, we cannot preload a load if the preload has to bypass a dependent store. For loops with definite short dependence distance[2] between particular memory references, we can choose to prefetch these loads into the cache. However, for some applications (e.g., sparse matrix computations), nested array indexes pose difficulties in obtaining exact dependence relations, and we do not want to discard the preload buffer support. For this situation, the *prestore* instruction and a *prestore log* (shown in Figure 6.7) are added to the

[2]We define short dependence distance as a number of processor cycles that is less than the memory latency.

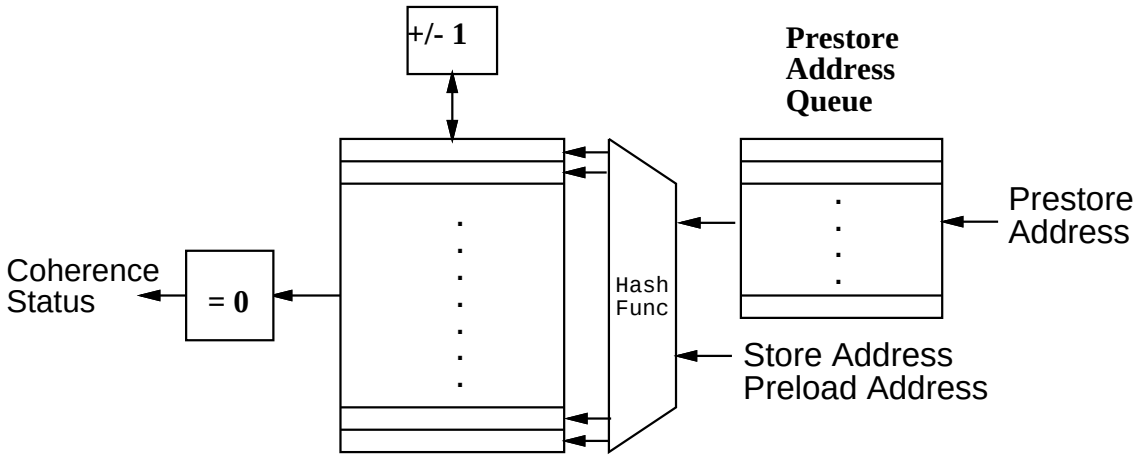


Figure 6.7 Design of a prestore log to maintain coherence.

architecture. These additions allow the preload to be converted into a cache load (prefetch) if the preload address may conflict with a later store address.

During compilation, prestores are generated along with preloads in sequential order. The prestore instruction contains only information for the store address and not the store data. The prestore instruction is a hint to the preload buffer that there might be a later store (storeb) that can affect the correctness of the data preloaded. Prestore does not make a memory request; it affects only the contents of the prestore log. Three types of addresses enter the prestore log: prestore, storeb, and preload. All addresses entering the prestore log are hashed to access an entry. The prestore instruction adds one to the prestore log entry while the storeb instruction subtracts one. Therefore, each entry contains the number of stores that may write to the same address as a later preload that hashes into the same entry. The preload instruction checks the prestore log entry before it sends the memory request to the system bus. If the log entry content is greater than zero, a later store may affect the preload memory contents. Therefore, the preload request is modified to a cache load, and the corresponding C flag is set. When the load request for the preload data sees that the entry is unavailable due to a coherence problem, the load request is routed to the data cache.

Instructions	Prestore Log Entry	Coherence Status
	0	
prestore 0(60)	1	
prestore 0(20)	2	
preload 0(60)	2	1
preload 0(20)	2	1
store 0(60),r5	1	
storeb 0(60)	1	
preload 0(60)	1	1
store 0(20),r3	1	
storeb 0(20)	0	
loadb r8,0(60)	0	
loadb r7,0(20)	0	
preload 0(20)	0	0

Figure 6.8 Example coherence solution.

Figure 6.8 is used to clarify the concept. Addresses 60 and 20 hash into the same prestore log entry. When prestore 0(60) and prestore 0(20) obtain the bus, no memory request is made. The addresses 60 and 20, however, are hashed into the prestore log and the content of the entry is appropriately increased. When later preloads execute, the same addresses hash into the same entry. The prestore log notes that the content is greater than zero, and the coherence status is set to one. The preload request is then aborted, and a cache load request is sent instead. Note that even though the second preload from address 60 can obtain the correct memory content, the preload is aborted due to mapping conflicts. The associated loadb instructions for addresses 20 and 60 will be redirected to the cache since their respective C bits are set. The prestore log can be made less conservative by increasing the number of log entries to decrease the chance of mapping conflicts. The last preload of address 20 will operate normally.

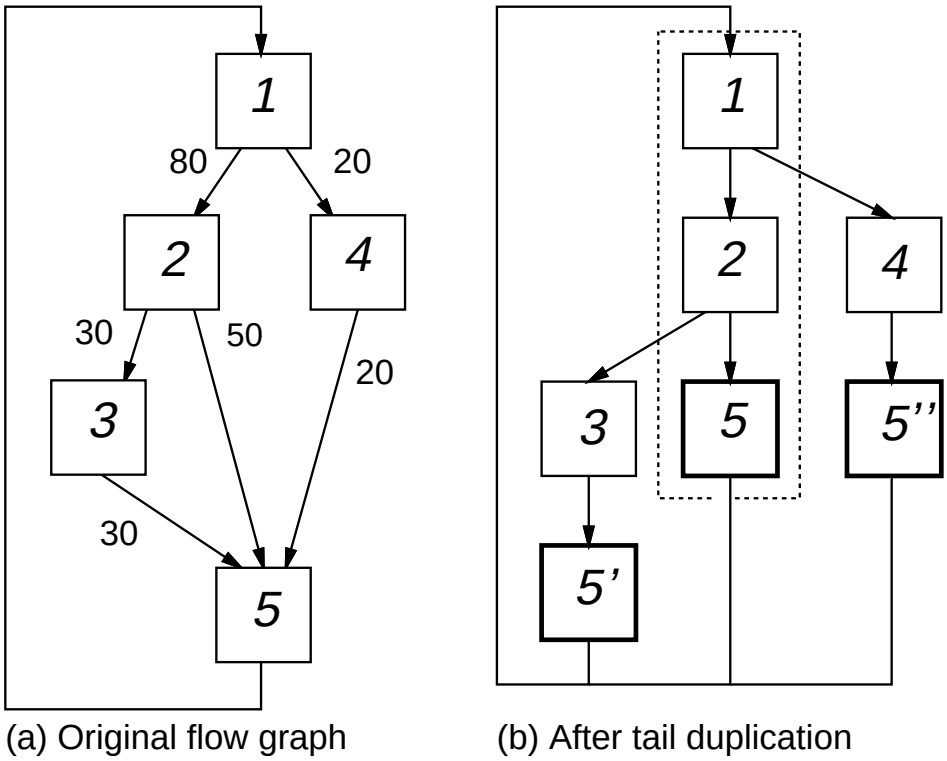


Figure 6.9 Example flow graph with conditionals.

Like the preload buffer, the number of bits in each prestore log entry is an architectural parameter that must be made known to the compiler. This is to prevent overflow of the prestore log entries.

6.3.4 Dealing with conditionals

All load instructions and their preload data entry locations are sequential as long as the same sequence of memory loads is executed for all loop iterations. This may not be true, however, for memory loads within conditional statements. Three solutions are provided: no prefetching, cache prefetching, or perform preloading with the help of the *skip* instruction. This section will concentrate on the third solution.

Consider the flow graph in Figure 6.9(a). The numbers beside the flow arcs are the execution frequency of that particular path of control. The execution frequency is obtained through

program profiling. As shown in the figure, there is more than one path through the loop, and this may cause a nonconstant number of loads to be executed for each loop iteration. The preload buffer, however, assumes that this number is constant so that it can be stepped through sequentially. Using the profile information, however, an important path of execution can be identified. In this case, they are blocks 1, 2, and 5, which is called the main path. All other paths from the exits of the main path are the side paths. By performing tail duplication, a single path of control is extracted; the resulting control flow is shown in Figure 6.9(b) and the resulting code is shown in Figure 6.10(a). Loop unrolling is performed to increase the scheduling scope. Preloading is then performed on the resulting program structure.

Code scheduling and preloading are first performed for the main path (Figure 6.10(b)). This involves the insertion of preloads and the transformation of the appropriate loads into loadbs. The next step is to fix the side path. The simplest algorithm is to duplicate any preloads after the individual exits of the main path into individual side paths and leave all memory accesses in the side paths as cache loads. As shown in Figure 6.10(c), all preloads in blocks 2 and 5 are duplicated into blocks 4 and 5", and all preloads in block 5 are duplicated into blocks 3 and 5'. This ensures that the preload buffer is maintained consistently regardless of the execution path taken through the loop.

Optimizations can be performed to utilize the available data in the preload buffer for the side paths. Since the ordering of the loadbs within the main path is now fixed, reading of the preload buffer from the side path can be emulated using the *skip* instruction. Skip allows the preload buffer to jump over preloads which are not read in the side paths. Before loading from the preload buffer, the skip instruction increments the outgoing data counter to the correct entry value (Figure 6.10(c)).

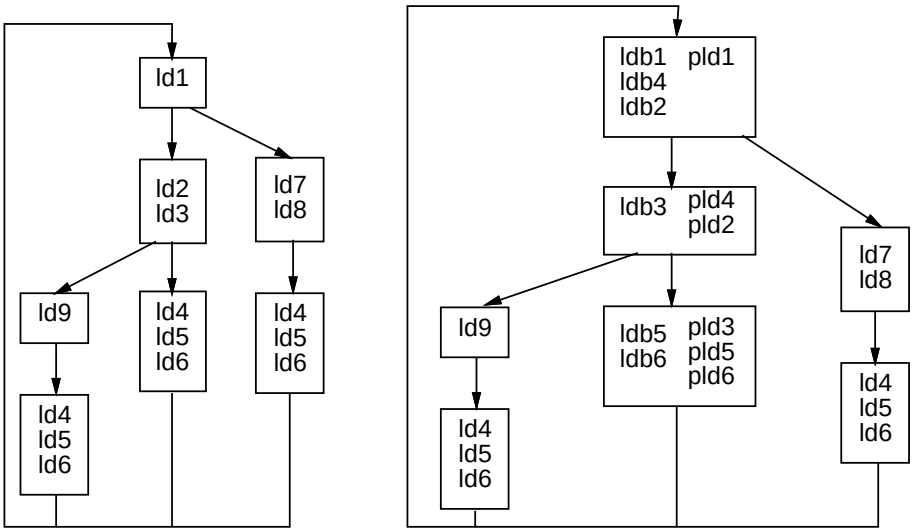


Figure 6.10 Example of preloading for conditionals.

6.3.5 Multiple preload buffer accesses

Superscalar processors can potentially make multiple load requests per cycle. Interleaving the cache banks is one way to reduce the access conflicts. If more than one reference is competing for the same memory bank, it is typical to allow only one to proceed while stalling the others. This bank mapping conflict, however, does not occur for the preload buffer. The preload buffer returns requests from sequential entries. By allowing a number of interleaved banks to be equal to the maximum number of allowable loadbs per cycle, the preload buffer can anticipate the loadb requests and store them in the preload buffer output latches. The loadb request performs only a latch-to-register transfer which can be accomplished in minimal cycle time. Exceptions occur only if the valid bit is not set or the coherence bit is set. In this case, the requester either waits for the data to return or sends a new memory request to the data cache.

6.3.6 Context switch

When a processor switches context during the execution of a preloaded loop, the preload buffer states must be saved to ensure correctness when the process resumes execution. One option is to save the entire preload buffer state. This may not be feasible when the number of entries within the preload buffer is large. The proposed solution is simply to save the outgoing data counter and the incoming address counter (Figure 6.4). All preload requests before the context switch should be prevented from returning to the preload buffer. When returning to the context, the counter values are read back, and the C bits of all of the entries from the outgoing data counter up to the outgoing address counter are set. The incoming data counter and the outgoing address counter values are initialized to the incoming address counter. The coherence bit of the preload buffer ensures that correct data are fetched for later loadb accesses.

6.4 Algorithm

To study the problem with cache prefetching, we have implemented a prefetch/preload algorithm in the IMPACT compiler which supports both scalar and superscalar compilation [28]. The prefetch instruction is inserted at the assembly level a number of cycles ahead of the corresponding load instruction within an inner loop to cover the memory latency. A simple cache profiler is used to determine the miss rate of a particular memory access for a given cache size. A 1K cache is used for all of the cache profiling done in this chapter. By using the miss rate information, unnecessary prefetches to data that tend to be in the cache can be eliminated. References with a nonzero miss rate will be prefetched for this chapter, and these loads are marked by the compiler. While generating the cache misses using the profiler, the stride of each memory reference is recorded. If the stride of an access is equal to the access size, a prefetch instead of a preload is generated to utilize the sequentiality of the cache better. The stride information is also used for later optimizations.

First, the loop is scheduled to determine the number of cycles required to execute under ideal memory conditions. Dividing a given memory latency by this cycle time, the compiler uses the ceiling of the result to decide the number of loop iterations to prefetch ahead. Next, the instructions generating the load address offsets are marked within the loop. If the size of the structure being accessed is statically declared and the offset producing instructions have loop invariant increments, the stride information from the cache profiler can be used. By using the stride information, many of the address calculations for prefetching can be eliminated. Abstract interpretation can be used by the compiler to determine the value of an invariant access stride. However, we chose to use profile information to reduce the compiler complexity. If the profiled access stride can be used, a prefetch/preload instruction is inserted for each array reference. For

example, the prefetch instruction for *load destination,(base_addr + int_offset)* can be generated as *prefetch (base_addr + int_offset + stride * prefetch_iteration)*. The *base_addr* is the address calculated based on each loop iteration. If the profiled access stride cannot be used, the marked address calculation instructions are duplicated into the loop preheader and also into the loop body. These instructions are copied n times into the loop preheader based on the prefetch iteration number. Also, prefetches (or preloads) are inserted into the loop preheader and loop body. For preloads, the corresponding loads are changed into loadbs. Register renaming is performed for the duplicated instructions.

Care must be taken when marking the loads for preloading. If the address calculation is dependent on the direction of a conditional within the loop, then preloading may not be correct. In this case, prefetch is generated instead of a preload. Also, if a load address is dependent upon another load instruction, the address producing load must also be prefetched(preloaded). Therefore, the prefetch algorithm is a two-step process; the first pass is used to catch the simple loads, and the second pass is used to catch the address producing loads.

Care also must be taken when memory dependences exist within and between the loop bodies. If an intermediate store exists between a preload and the corresponding loadb, the store instruction is also marked as an address producing instruction. When code duplication occurs, the store instruction is transformed into a prestore, and the corresponding store instruction is changed into a storeb instruction.

6.5 Experimental Evaluation

Experiments were conducted to evaluate the performance of preloading. The performances of cache prefetching and no cache prefetching were also measured for comparison. First, the

miss ratio and bus traffic are shown when employing preloading. Then detailed simulations are done for two cache sizes. Finally, the cache block size is decreased to measure the performance variation.

6.5.1 Simulation architecture

The base architecture is an in-order superscalar processor capable of fetching four instructions per cycle. Nontrapping hardware is assumed to suppress exceptions caused by preload and prefetch instructions. The functional units are pipelined and any combination of instructions can be dispatched per cycle. The instruction latencies are the same as those used in Section 6.2. The primary cache size varies according to the simulation. In all cases, the primary cache is assumed to be nonblocking with an infinite-sized miss queue. The second-level cache is fixed at 1MB and is assumed to be blocking. Both caches are direct mapped and employ write through. An eight-entry write buffer is provided. The penalty for a first-level cache miss is 10 cycles to access the second-level cache. If the second-level cache misses, an additional 20 cycle penalty is incurred to access main memory. The memory latency for prefetching is therefore a total of 30 cycles. The address bus is fully pipelined and can take one memory request every cycle. The data bus bandwidth, however, is limited to transfer a double word every cycle. Therefore, the data transfer for a 32-byte cache block will occupy the data bus for four cycles.

6.5.2 Miss ratio and bus traffic

During compilation to support preloading, the maximum number of preload buffer entries used for each benchmark was recorded. Table 6.2 lists the benchmarks and their preload requirements. Benchmark gmtry requires the greatest number of entries. A maximum of 64 buffer entries were used for all later simulation results. To allow for the size of the preload

Table 6.2 Maximum number of preload buffer entries used.

Benchmark	*Buffer Entries*
matrix300	32
fft2d	16
cholsky	32
btrix	24
gmtry	44
vpenta	32
tomcatv	34

Table 6.3 Comparison of normalized miss ratios.

	No Prefetch			*With Preload*		
Bench	*4KB*	*8KB*	*16KB*	*2KB*	*4KB*	*8KB*
mat300	1.00	0.80	0.57	0.03	0.02	0.02
fft2d	1.00	0.93	0.86	0.06	0.05	0.05
cholsky	1.00	0.94	0.75	0.04	0.04	0.03
btrix	1.00	0.64	0.33	0.18	0.12	0.09
gmtry	1.00	0.97	0.75	0.02	0.00	0.00
vpenta	1.00	1.00	1.00	0.05	0.05	0.05
tomcatv	1.00	0.37	0.25	0.08	0.02	0.02

buffers, simulations using preloading assume that data cache size is half the size of simulations with no preloading.[3] Thus, a simulation supporting prefetch with an 8K cache is compared to a simulation supporting preloading with a 4K cache.

First, the performance of the preload buffer was characterized by its miss rate reduction. Tests were run using 32-byte blocks with no preloading for 4, 8, and 16KB caches. In all experiments, a preload data element that has not been validated before its associated load occurs is counted as a miss. The normalized miss ratios for preloading are shown in Table 6.3. When no preloading is performed, an increase in the cache size does not significantly reduce

[3] A 64-entry preload buffer requires 64 * (3 * 4 bytes) = 768 bytes plus the valid and coherence bits.

the number of misses for several of the benchmarks. By performing preloading, a significant percentage of misses are eliminated for all benchmarks.

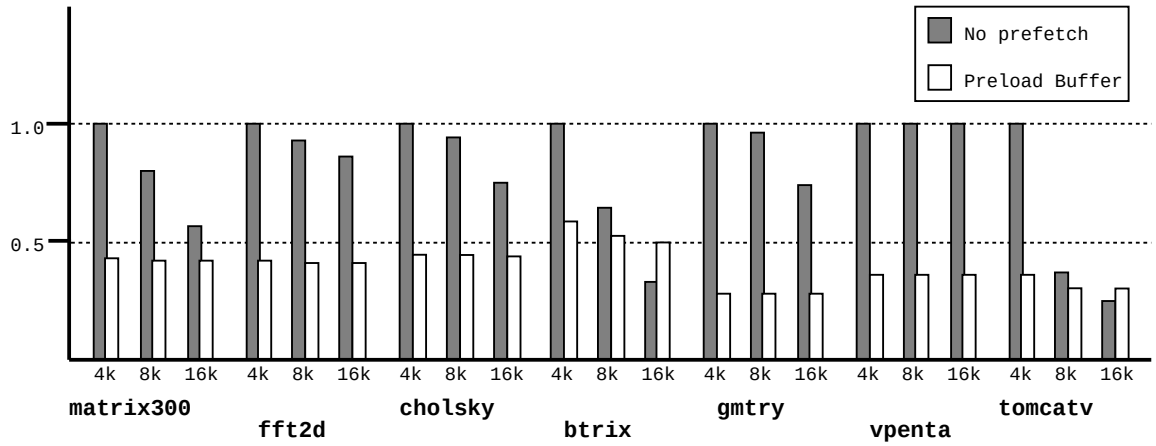


Figure 6.11 Comparison of normalized bus traffic.

Next, the bus traffic was evaluated. The normalized bus traffic ratio comparing no preload versus preload is shown in Figure 6.11. Surprisingly, preloading reduces the bus traffic for all but two cases. Since the miss ratio is always reduced, a decrease in bus traffic usually results in an improvement in system performance. Detailed processor simulation is required to show the extent to which preload improves the overall system performance. Also, in the case of *btrix* and *tomcatv*, we must find out how the increase in bus traffic affects the performance of preloading with detailed simulation.

6.5.3 Detailed simulation

Detailed system simulation was performed to account for the resource constraints of the memory system. The comparison of cache prefetch was included as an alternative solution to preloading. The execution cycle for each simulation is divided into two parts: the instruction execution time and memory overhead. The instruction execution time is the time required to execute the benchmark if all references can be satisfied by the primary cache. The memory

overhead is the time that the processor is waiting for the data to return from the second level cache or the main memory. The discrepancy between the instruction execution time is due to prefetch and preload overhead.

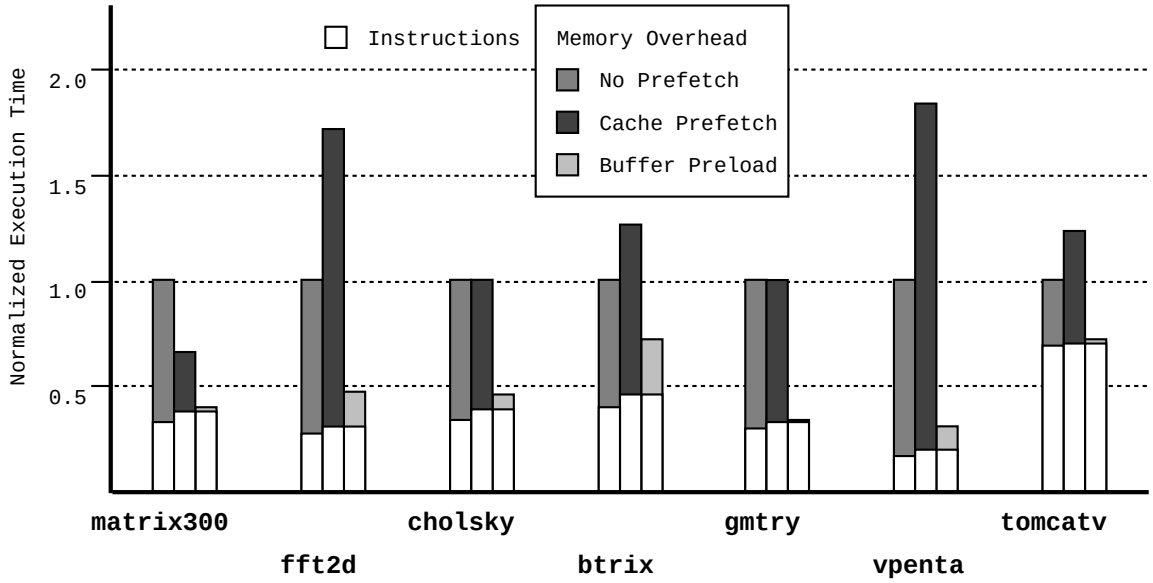


Figure 6.12 Comparison of normalized execution time for 8K cache.

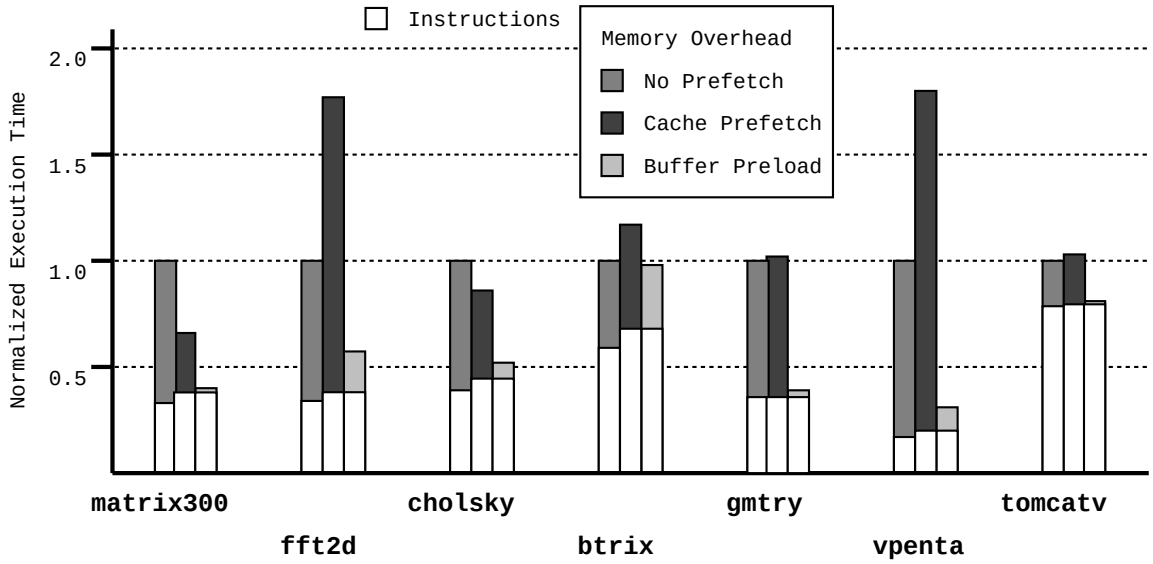


Figure 6.13 Comparison of normalized execution time for 16K cache.

The 8K and 16K cache results are shown in Figure 6.12 and 6.13, respectively. It is not surprising that memory overhead accounts for a large portion of the execution time in an four-

issue superscalar processor. Also, due to the increase in bus traffic, cache prefetching improves the performance of these benchmarks only in the case of *matrix300* and *cholsky*. Preloading always performs better. However, it is interesting to note that where preloading increases the bus traffic (*btrix*), the benefit of preloading drops significantly. It is imperative to eliminate misses while reducing the bus traffic to guarantee an improvement in system performance.

6.5.4 Effects of cache block size

Up to this point all of the cache simulation assumed 32-byte blocks. In this section, the cache block size is restricted to a double word to eliminate the effects of sequential prefetch. The small block size prevents any unused data from being placed in the cache. A small block size is beneficial in reducing cache pollution for long stride accesses. However, a small block size creates extra bus traffic which degrades system performance for short stride accesses.

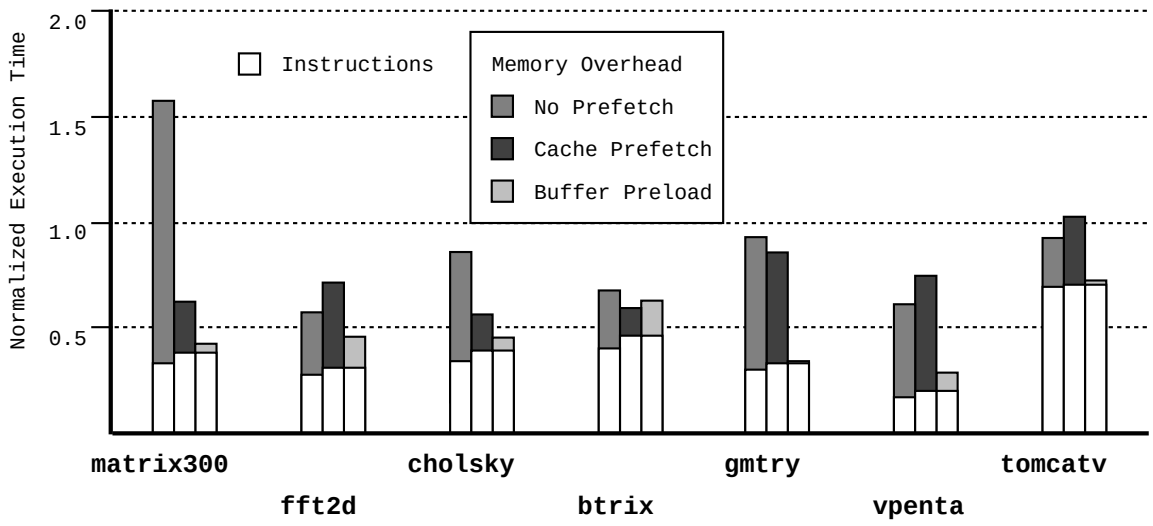


Figure 6.14 Comparison of normalized execution time for 8K cache with 8-byte blocks.

In Figure 6.14, the results for an 8K cache are shown. All results are normalized against the result of no prefetch using an 8K cache with a 32-byte block. A noticeable performance shift can be seen for no prefetch and cache prefetch. *Matrix300* performance dropped significantly

due to the loss of the benefit of long cache block fetches. The elimination of unnecessary cache block fetches greatly improved the performance of the other benchmarks. However, the benefits of cache prefetch are still unpredictable due to conflicts between the current working set and the future working set. As a result of these conflicts, the effective cache size may be smaller than its physical size depending on the mapping of the structures into the cache. These conflicts can also remove necessary data from the cache, and their subsequent reloading results in increased bus traffic.

It is interesting to note that the improved cache prefetch performance with *btrix* makes cache prefetch perform slightly better than the preload buffer. In comparison to Figure 6.12, the main observation is that the performance of the preload buffer stays relatively constant. This predictability is a major advantage of the preloading approach.

6.6 Summary of Preload Buffer

This chapter proposes an architectural support, the preload buffer, to cooperate with the compiler in tolerating long memory access latencies. Unlike previously proposed methods of nonbinding cache loads, a preload is a binding access to the memory system. The prefetch buffer requires associative searches and a coherence mechanism which is high in cost and access time. Also, we have shown that cache prefetching can produce unpredictable performance results due to cache interference among data arrays. The preload buffer eliminates some of the problems caused by compiler-assisted cache prefetching either into the cache or into a prefetch buffer.

The main feature of the preload buffer is expandability and simplicity. The benefits of the preload buffer are also predictable. The problems of pollution and dimensional conflicts due to cache prefetch are eliminated. The preload buffer assumes no searching or comparisons.

Access to the preload data can be made faster than access to the data cache. With simple interleaving, accesses to the preload buffer are independent of the access pattern and processor issue rate, and are therefore memory bank conflict free. The preload method will default to cache prefetching in the event of updates to the cache which invalidate preloaded data.

With advanced compiler technology using IMPACT, the preload buffer is shown to be effective at hiding long memory access latencies. In particular, preloading is shown to achieve better performance than cache prefetching for a set of benchmarks. In all cases, preloading decreases the bus traffic and reduces the miss rate when compared with no prefetching or cache prefetching. Overall, the preload buffer is a promising concept and deserves further research.

CHAPTER 7

CONCLUSIONS

7.1 Summary

This thesis has presented schemes which allow instruction reordering in the presence of memory dependence. Data preloading is a combination of architectural and compiler support which allows a dependent load to be moved above an ambiguous store. Correctness of execution is maintained by both the hardware and software. Data preload is divided into two major categories: 1) only the loads are allowed to move above ambiguous stores, and 2) both loads and flow-dependent instructions of the load are allowed to move above ambiguous stores. Depending on how much distance is between the moved load and its flow dependent instruction, different amounts of memory latency are hidden from the view of the pipeline. Also, as flow dependent instructions move along with the load, more freedom in instruction reordering is achieved, thereby uncovering more instruction-level parallelism.

Experimental evaluation of the proposed methods has shown that memory dependence can limit instruction reordering. By allowing memory dependence to be removed, benefits start to surface in four-issue processors with the most benefits seen in eight-issue processors. Limitations to the proposed schemes include the following: 1) increased dynamic instruction count which requires higher instruction bandwidth to compensate, 2) difficulty involved in static branch prediction which can result in optimization of the wrong branch path, 3) nonrobust integrated register allocation and instruction scheduling algorithm which can cause excess register spilling

when more instructions are scheduled in parallel, 4) limited scheduling scope with the superblock to extract more parallelism, and 5) existing memory dependence analyzer has already removed most of the critical memory dependences and the program cannot benefit from the proposed schemes.

Experimental analysis is conducted for the proposed schemes. It was shown that the hashing memory conflict buffer design is cost effective, and a 64-entry memory conflict buffer is sufficient to uncover most of the parallelism for a set of benchmarks. It was also shown that cache pollution is a major drawback of cache prefetching and preload buffer support can complement cache prefetching for latency tolerance. To cover thirty cycle of memory latency, a 64-entry preload buffer is adequate.

7.2 Future Research

As hardware real estate cost decreases and instruction issue bandwidth increases, the need for data preload support will grow. Heuristics for ambiguous memory dependence removal may not be as good as one can hope for. The use of memory dependence profiling can give a static weight of memory address matches. With a dynamic estimate of memory address matches, the compiler can make more intelligent decisions for ambiguous memory dependence removal. Hyperblock [5] scheduling provides a larger scheduling scope than does superblock, and it is this larger scheduling scope that data preload can utilize to extract more instruction-level parallelism. As instruction-level parallelism increases, ways to increase the effective number of physical registers must be introduced to compensate for the ever-growing need for more registers. Although the memory dependence analyzer can never replace data preload support, a better memory dependence analyzer can eliminate overhead introduced by preload supports by

actually guaranteeing the independence of memory instructions. Further evaluation of preload support would include modifying the compiler.

REFERENCES

[1] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, Nov. 1992.

[2] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," *IEEE Transactions on Computers*, Dec. 1992.

[3] V. A. Guarna Jr., "Analysis of C programs for parallelization in the presence of pointers," M.S. thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, Illinois, 1987.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 296–310, June 1990.

[5] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated-execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[6] M. J. Wolfe, "Optimizing compilers for supercomputers," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1982.

[7] J. R. Allen, "Dependence analysis for subscripted variables and its application to program transformation," Ph.D. dissertation, Department of Mathematical Science, Rice University, 1983.

[8] U. Banerjee, *Dependence Analysis for Supercomputing.* Boston, MA: Kluwer Academic Publishers, 1988.

[9] H. Dietz and C. H. Chi, "Cregs: A new kind of memory for referencing arrays and pointers," in *Proceedings of Supercomputing '88*, pp. 360–367, Nov. 1988.

[10] B. Heggy and M. L. Soffa, "Architectural support for register allocation in the presence of aliasing," in *Proceedings of Supercomputing '90*, pp. 730–739, Nov. 1990.

[11] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.

[12] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Comparing dynamic and static code scheduling for multiple-instruction-issue procesors," in *Proceedings of 24th Annual Workshop on Microprogramming and Microarchitectures*, (Albuquerque, NM.), Nov. 1991.

[13] P. G. Emma, J. W. Knight, III, J. H. Pomerene, R. N. Rechtschaffen, and F. J. Sparacio, "Posting out-of-sequence fetches," Feb. 1991. United States Patent No. 4991090.

[14] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of 8th Annual International Symposium on Computer Architecture*, (Minneapolis, MN), pp. 81–87, May 1981.

[15] J. E. Sicolo, "A multiported nonblocking cache for a superscalar uniprocessor," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1992.

[16] R. L. Lee, P. C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors," in *Proceedings of 16th International Conference on Parallel Processing*, pp. 28–31, Aug. 1987.

[17] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessor with memory hierarchies," in *Proceedings of International Conference on Supercomputing*, (Amsterdam, The Netherlands), pp. 354–368, June 1990.

[18] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of Supercomputing '91*, pp. 176–186, Nov. 1991.

[19] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40–52, Apr. 1991.

[20] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of 24st Annual Workshop on Microprogramming and Microarchitectures*, (Albuquerque, NM.), Nov. 1991.

[21] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proceedings of 18th Annual International Symposium on Computer Architecture*, (Toronto, Canada), pp. 54–63, June 1991.

[22] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *Proceedings of 18th Annual International Symposium on Computer Architecture*, (Toronto, Canada), pp. 43–53, May 1991.

[23] T. C. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 87–106, 1991.

[24] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 51–61, Oct. 1992.

[25] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of 25th Annual Conference on Microprogramming and Microarchitectures*, (Portland, Oregon), Dec. 1992.

[26] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62–73, Oct. 1992.

[27] P. P. Chang, "Compiler support for multiple-instruction-issue architectures," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1991.

[28] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of 18th Annual International Symposium on Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.

[29] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," in *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[30] H. S. Warren, Jr., "Instruction scheduling for the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, pp. 85–92, Jan. 1990.

[31] R. M. Russell, "The Cray-1 computer system," *Communications of the ACM*, vol. 21, pp. 63–72, Jan. 1978.

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

[33] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems.*, (Palo Alto, CA), pp. 180–192, Oct. 1987.

[34] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of ACM SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105, 1982.

[35] W. Y. Chen, S. A. Mahlke, N. J. Warter, R. E. Hank, R. A. Bringmann, S. Anik, D. M. Lavery, J. C. Gyllenhall, T. Kiyohara, and W. W. Hwu, "Using profile information to assist advanced compiler optimization and scheduling," in *Advances in Languages and Compilers for Parallel Processing*. Cambridge, MA: MIT press, 1992.

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

[37] 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," Tech. Rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, Dec. 1991.

[38] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of 14th Annual Workshop on Microprogramming and Microarchitectures*, Oct. 1981.

[39] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of 1989 ACM Conference on Programming Language Design and Implementation*, (Atlanta, Georgia), June 1988.

[40] W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Tolerating first level memory access latency in high-performance systems," in *Proceedings of 21st International Conference on Parallel Processing*, Aug. 1992.

VITA

William Yu-Wei Chen was born on July 6, 1965, in Taipei, Taiwan. He received his B.S. degree in electrical engineering from The Ohio State University. From 1988 to 1990, he was a teaching assistant in the Electrical and Computer Engineering Department at the University of Illinois, Urbana-Champaign, where he received his M.S. degree. Since 1990, he has been a research assistant in the Center for Reliable and High-Performance Computing, also at the University of Illinois. In August 1993, he joins Intel Corporation in Santa Clara, CA, working on the next generation superscalar compiler.