Tolerating Data Access Latency with Register Preloading

William Y. Chen Scott A. Mahlke Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, Illinois, 61801.

Tokuzo Kiyohara

Media Research Laboratory
Matsushita Electric Industrial Co., Ltd.
Osaka, Japan

Pohua P. Chang

Intel Corporation
Hillsboro, OR 97124

Abstract

By exploiting fine grain parallelism, superscalar processors can potentially increase the performance of future supercomputers. However, supercomputers typically have a long access delay to their first level memory which can severely restrict the performance of superscalar processors. Compilers attempt to move load instructions far enough ahead to hide this latency. However, conventional movement of load instructions is limited by data dependence analysis. This paper introduces a simple hardware scheme, referred to as *preload register update*, to allow the compiler to move load instructions even in the presence of inconclusive data dependence analysis results. 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.

Keywords: data dependence analysis, load latency, register file, register preload, VLIW/superscalar processor.

1 Introduction

In order to increase performance, future supercomputers can utilize superscalar processors to exploit fine grain parallelism inherent to applications. Due to the memory requirements of many supercomputer applications, the first level memory is usually large with a high access latency. 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 [1]. [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.

Data dependence analysis determines the relation between memory references. Three possible conclusions can be reached regarding the relation between a pair of memory references: they always access the same location, they never access the same location, or they may access the same location. In the first two cases, the compiler can utilize this information to optimize and schedule the reference pair. In the third case, the inconclusive result typically disables optimizations and code reordering. For example, consider the scheduled code segments in Figure 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 1a. 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 1b. This problem compounds as the processor is able to issue more instructions per cycle, since each cycle that the processor has to wait for memory references becomes more significant to the

[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 1: Problem with data dependence.

overall execution time.

For array references, many algorithms exist to perform data dependence analysis [2] [3] [4]. However, there are many cases where these data dependence analysis algorithms cannot provide conclusive results [5] [6]. Due to possible reference conflicts, a dependence must be assumed between the reference pair to ensure correct program execution. Furthermore, programming languages which allow data types, such as structures and pointers, pose even more difficulties for data dependence analysis [7] [8]. With preload register update, the dependence between a load/store pair is removed regardless of the dependence relation, and a more compact schedule can thus be achieved.

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 [9]. Using a hardware monitor as proposed by Emma *et al.*, loads can bypass stores even when the store addresses are unknown [10]. 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.

A combined hardware and compiler scheme to keep in register a value that can be accessed via multiple variable names, or aliases, has been proposed [11]. 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. This approach can also be used to reorder dependent load/store pairs when the load destination and the store value reside in the same alias set. However, the effectiveness of this approach is limited by the size of the alias sets provided in the hardware implementation. To remove this constraint, a separate linked list of registers for each alias can be maintained during execution [12]. Due to the possibility of searching for the leader of the linked list, though, register access time may require an extra cycle.

Dependent load/store pairs can be reordered by inserting explicit address comparison and conditional branch instructions during compilation [13]. Instructions are also inserted to repair for the incorrect execution of wrongly reordered reference pairs. The extra instructions, however, can cause a large execution overhead as a result of aggressive code reordering.

In this paper, 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 2, 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 3. In Section 4, the effectiveness of preload register update is evaluated for a set of non-numeric and numeric benchmark programs.

2 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 3.

2.1 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 register if the preload and the store reference the same memory location. Figure 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 2.3). The coherence mechanism will

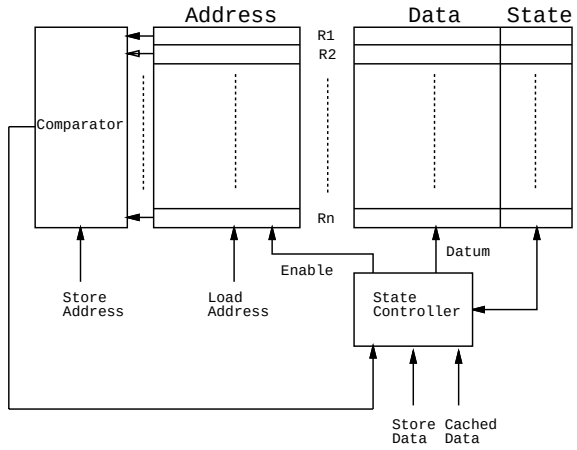

Figure 2: Overview of preload register update.

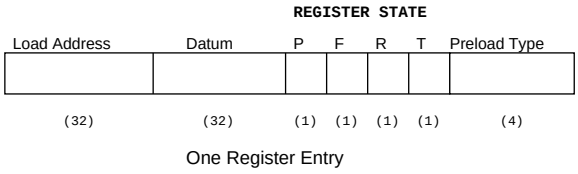

Figure 3: Register states implementation detail.

continue to operate until a commit instruction is executed or the register entry is redefined by a normal instruction.

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 is presented in Figure 3. The opcode type of the preload is encoded and saved in the preload type field. This is used for data alignment and 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 state (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 state (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). Since if the preloaded value is not used, the trap can be ignored. Detection of exceptions for optimized and scheduled code is discussed in [14]. In this paper, we focus on the use of the preload register to improve the overall program performance.

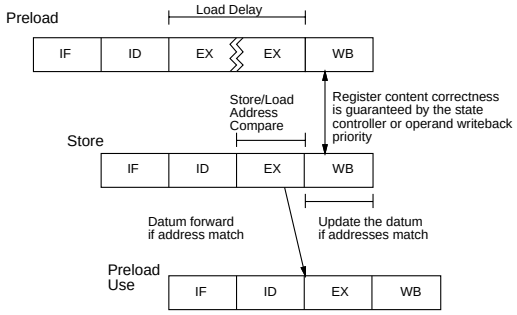

Figure 4: Example pipeline stage and instruction relation assuming load latency of 2.

2.2 Implementation Timing and Pipeline Stages

The stages of the pipeline model are illustrated by an example in Figure 4. We wish to demonstrate two issues with this example: register content updating and data forwarding 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 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 1. The associated state diagram is presented in Figure 5 while the possible inputs used in the state diagram are given in Table 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 state $PF\overline{R}$ 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 simple interlocking mechanism similar to that used in CRAY-1 [16]. If the freeze state is immediately known at the time of the 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

[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 anti dependences are handled by register renaming at the decode stage [15]. In this case, the coherence mechanism operates on the physical registers.

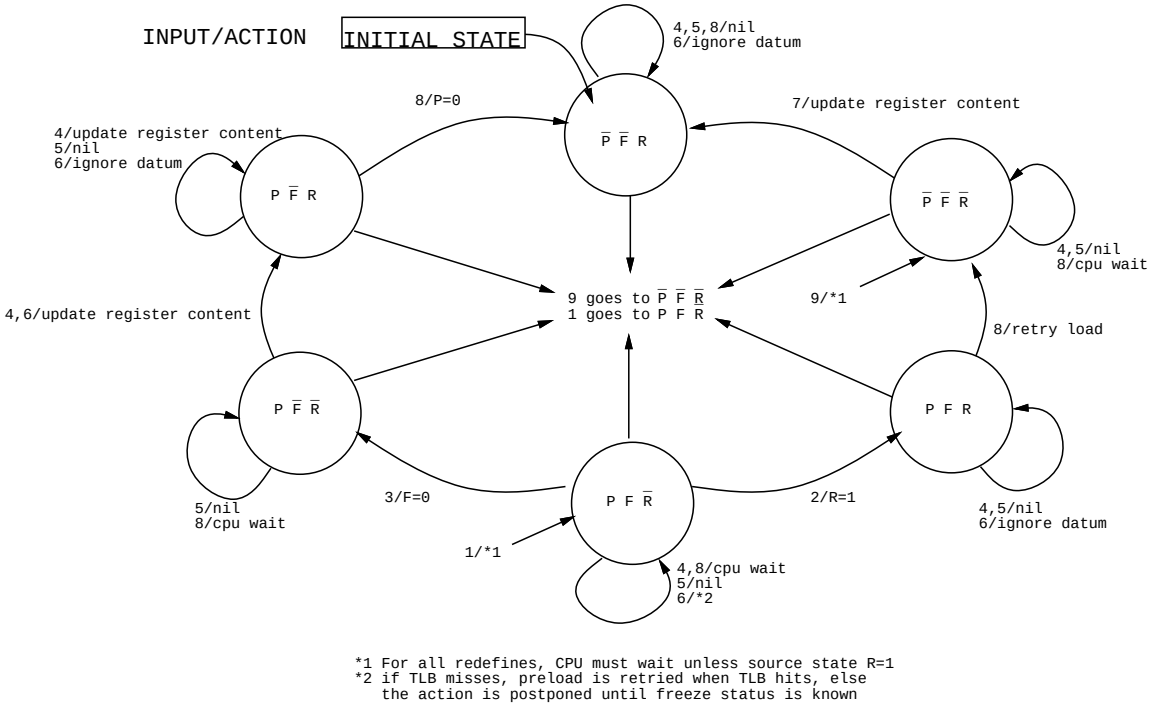

Figure 5: State diagram of preload register controller.

Register State			*explanation*
P	*F*	*R*	
0	0	0	Register redefine by a non-preload instruction, not ready
0	0	1	Register contain 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 1: Explanation of register states.

Type	*Explanation*
1	Instruction is a preload
2	Preload needs to be frozen
3	Preload does not need to be frozen
4	Store addr matches the preload addr
5	Store addr does not match preload addr
6	Preload datum is delivered
7	Non-preload datum is delivered
8	Preload datum use, commit
9	Reg is defined by an instr other than a preload

Table 2: Input types to the preload register controller.

lag time before the result is available, therefore state 110 is included. State 001 is the initial state of all registers.

2.3 Committing Preload Data

After the execution of all 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

in Section 3, the compiler does not move the use of 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 artificial use of the register by performing a *move* to a register hardwired to 0 (such as R0 in the MIPS R2000 [17]) 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.

2.4 An Example of Preload Register Update Operation

We illustrate preload register update with an example. Figure 6a shows a load/store pair whose address 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 6b). 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 6c). 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.

2.5 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 all sized register files.

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 $m < n$); m address registers each with a general purpose register pointer (GRP) field and a valid bit (V) added (see Figure 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 needs to be kept coherent for all subsequent memory stores. If there is more than

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

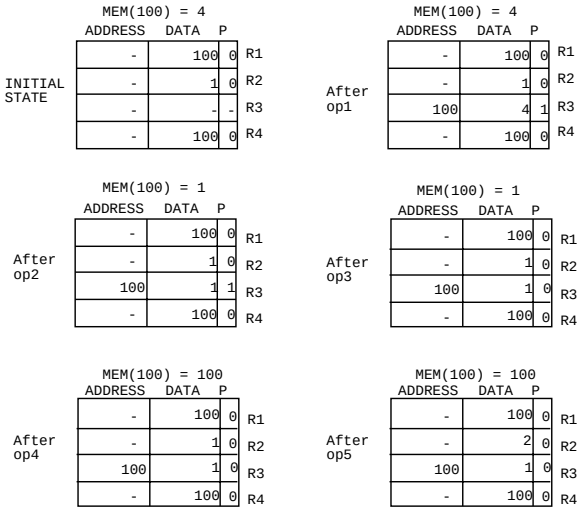

Figure 6: An example of preload register update.

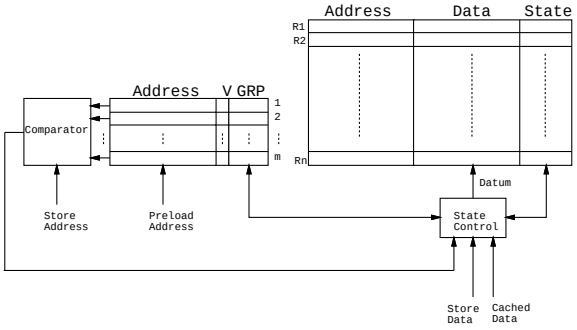

Figure 7: A Subset Design of Preload Register Update.

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

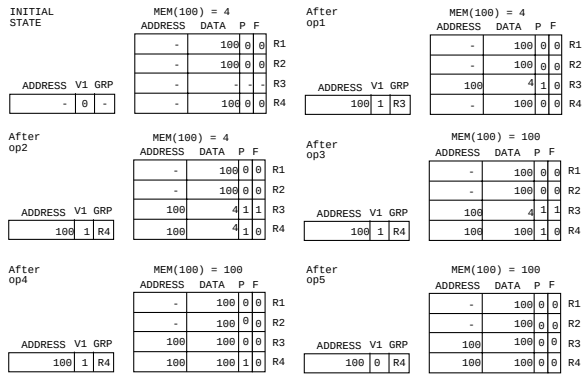

Figure 8: An example for preload in subset design.

one preload address entry which matches the store address, the pipeline is frozen and all matching registers are serially updated.

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 as to prevent stale preload addresses from occupying useful address register 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 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.

3 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 [1]; 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 only enter from the top but may leave from one or more exit points.

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

In order to take advantage of preload register update, the dependence graph construction phase needs 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 where 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 paper, the general code percolation model [1] is assumed, and non-trapping hardware [18] is used to suppress the exceptions caused by these preloads.

An example dependence graph for the code segment in Figure 10a is shown in Figure 9a. We assume a load latency of 2 cycles and a latency of 1 cycle for all other instructions 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 9b. By allowing the load to bypass the first store, the second ALU instruction

a) Before b) After

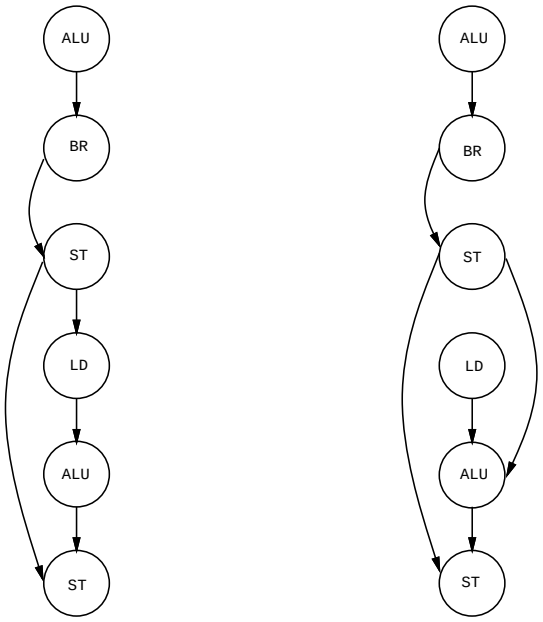

Figure 9: An example dependence graph.

can be scheduled earlier, thereby shortening the program critical path for a high issue rate processor. The resulting schedule is shown in Figure 10b. 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 10a would have taken 8 cycles to execute. However, the execution time of the code segment with preloading in Figure 10b remains at 5 cycles.

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 needs to be modified so that the preload destination registers tend not to be spilled before they are committed. This is accomplished by 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] In fact, only either the data register or the address register needs 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 needs to be saved, otherwise, the data register is saved.

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 10: Code scheduling and execution cycles.

4 Experiments

Compiler support for preload register update has been implemented based on the IMPACT-I compiler developed at the University of Illinois. The IMPACT-I compiler is geared towards high-performance scalar and superscalar processors. In this section, experimental results on the effectiveness of preload register update are reported for the twelve benchmarks listed in Table 3. The benchmark set consists of five numeric kernels and seven control intensive non-numeric programs. The benchmarks are divided into two groups according to their performance behavior that will be explained later. All benchmark programs are profiled with several different inputs. The profile information is used to identify superblocks in the benchmark programs.

4.1 Evaluation Methodology

To evaluate the performance of preload register update, each benchmark program is re-profiled using one input different from those with which it was originally profiled. Base on the new profile information, we derive the worst case execution time of each superblock for the instruction issue rates of 1, 2, 4 and 8. The worst case execution time is derived by considering the long instruction latencies that protrude from one superblock to another. To summarize performance results for a group, we report the harmonic mean of the speedup numbers of all benchmarks in that group. In the case of a cache miss, the pipeline is stalled, and all subsequent instructions cannot proceed until the cache miss is resolved. A blocking cache is simulated, therefore, all cache misses are serialized and are non-overlapping.

The base architecture for calculating all speedup numbers has an issue rate of one instruction per cycle and supports

Grouping	*Benchmark*	*Benchmark Description*
Group 1	cmp	compare files
	eigen	eigenvalues and eigenvectors
	espresso	truth table minimization
	gause	solve system of equations
	grep	string search
	sparse	solve sparse linear system
	wc	word count
Group 2	lex	lexical analyzer generator
	ludecom	LU decomposition
	matrix	matrix multiplication
	tbl	format tables for troff
	yacc	parser generator

Table 3: Benchmarks.

INT function	*latency*	*FP function*	*latency*
ALU	1	ALU	3
barrel shifter	1	conversion	3
multiply	3	multiply	4
divide	25	divide	25
load	varies	load (1 word)	varies
preload	varies	preload (1 word)	varies
store	1	store	1

Table 4: Instruction Latencies.

general code percolation. The instruction set is a superset of the MIPS R2000 instruction set [17] with extensions in branching capabilities. One branch delay slot that consists of N instructions for an N-issue processor is automatically allocated for 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. We assume CRAY-1 style interlocking and deterministic latencies (see Table 4) for all instructions except memory loads and preloads. 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.

4.2 Performance Evaluation

Ideal Cache Case

The full scale design of preload register update is evaluated here in terms of execution speedup with an ideal cache.

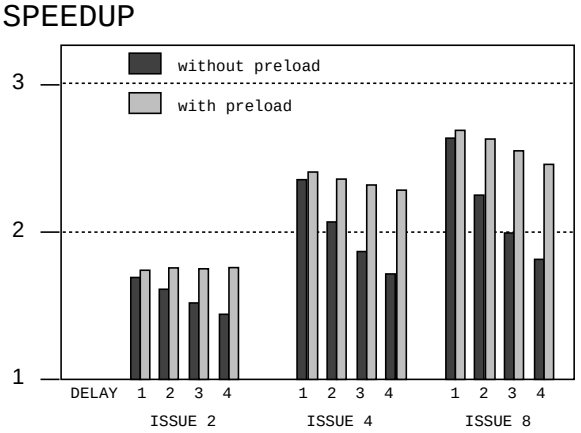

Figure 11: Speedup for Group 1 benchmarks.

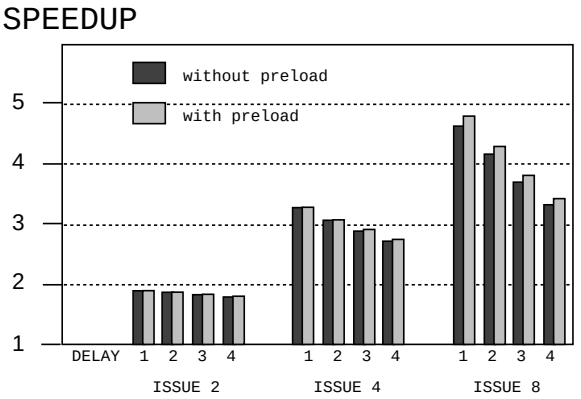

Figure 12: Speedup for Group 2 benchmarks.

We have divided the twelve benchmark programs into two groups. Group 1 contains the benchmarks that obtained substantial performance improvement with preload register update. Group 2 consists of the benchmarks which did not benefit significantly from preload register update. Figures 11 and 12 show the speedup achieved by superscalar processors over the base architecture with and without preload register update. We will examine the two groups separately and explain the difference.

First, the Group 1 benchmarks are able to tolerate the increased load latency better with preload register update. For example, the performance of an issue 4 processor with register preloading drops by only 5% when the load latency is increased from 1 to 4. However, without register preloading, the performance drops by 37%. Closer examination of the benchmark programs reveals that the scheduling of the Group 1 programs is limited by inconclusive data dependence analysis results. As a result, the extra freedom to reorder memory instructions provided by preload register update enables the superscalar processor to better tolerate longer load latency.

Second, the Group 2 benchmarks do not benefit from

Benchmark	*Max Preloads Used*
cmp	6
eigen	16
espresso	8
gause	15
grep	11
sparse	20
wc	2

Table 5: Maximum number of required preload registers in a subset design.

preload register update in general (see Figure 12). Examining the Group 2 benchmarks shows that the scheduling of these programs is not restricted by data dependence analysis. Also, a lack of stores in the critical region within these benchmarks provides more scheduling freedom for load instructions than Group 1 benchmarks. Group 2 programs, therefore, achieve a high level of performance without preload register update. For example, an issue 4 processor achieves more than 3 times speedup over the base architecture in Figure 12. These programs are examples where there are few opportunities for preload register update to further improve performance.

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

The characteristics of each benchmark determine the number of preload registers required. Also this determines the number of the address register entries required in the subset design. Table 5 presents the maximum simultaneously live preload registers for each of the Group 1 benchmarks. The address register requirement ranges from 2 to 20. Therefore for Group 1 benchmarks, 20 address registers are enough to achieve the performance level in Figure 11 for the subset design. The results for group 2 benchmarks are not given due to a lack of opportunities for preloading, and therefore the lack of need for preload registers.

Cache Miss Penalty

It is important to quantify the effect of cache misses on the overall performance, either with or without preload register update. Figure 13 shows the speedup of the Group 1 benchmarks when taking data cache miss penalty into account. The three bars associated with each load latency correspond to three different caches: ideal, 128K, and 64K. Each bar is divided into two sections, with and without preload register update. The cache is direct mapped with 32-byte blocks, and the cache refill latency is 50 cycles. Each data cache miss is assumed to cause the processor to

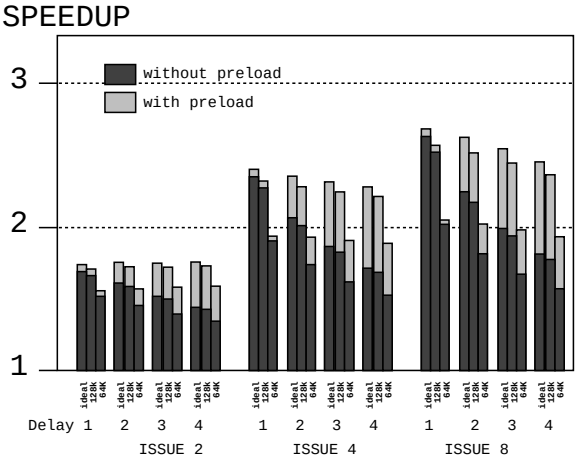

Figure 13: Speedup comparison under cache miss penalty for Group 1 benchmarks.

stall for the cache refill latency. Since data cache misses affect both the base scalar processor performance and the superscalar processor performance, speedup is calculated by taking data cache misses into account for both performance measurements.

We will first concentrate on the performance of register preload update under cache misses. As shown in Figure 13, preload register update maintains a relatively constant performance level across the load latencies shown for a given cache size. However, due to differing numbers of cache misses for various cache sizes, a higher performance level is obtained as the cache size increases. For 128K cache, the performance level is relatively unchanged with respect to the ideal case. The performance level for the 64K cache is noticeably lower than that of the 128K cache. This is mainly due to the large data working set of the numeric benchmarks (eigen, gause, and sparse). Thus, the result in Figure 13 illustrates the need to include data prefetching and other load latency hiding techniques in the compiler.

By comparing the result of with and without preload register update in Figure 13, there are two important observations. First, if doubling the 64K data cache causes an increase in load latency, then the performance increase is negligible without preload register update. However, with preload register update, the processor can effectively utilize the larger cache size to obtain higher performance even with the increased load latency. The performance thus achieved is approximately the same as if the cache access time has not increased. For example, across all issue rates, the performance improvement from a 64K cache with 1 cycle access time to a 128K cache with 2 cycle access time is negligible without preload register update. By adding the preload support, the 128K cache with 2 cycle access time achieves comparable performance as that of a 128K cache with 1 cycle access time. Second, preload register update becomes crucial as the load latency is increased. For higher load latencies (3 to 4 cycles), larger performance gains result

from providing preload register update with a 64K cache than increasing the cache size to infinite. Therefore, it is more important to support preload register update before an increase in cache size for higher load latencies.

5 Conclusion

This paper presents a detailed design of a hardware mechanism, referred to as preload register update. 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 is 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.

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. For programs whose scheduling is limited by inconclusive data dependence analysis results, preload register update achieves from 14% to 33% performance improvement for an issue 4 processor with load latency of 2 to 4 cycles.

This paper has focussed on the use of preload register update to assist code scheduling. There are many other optimizations within the compilation process which can benefit from this hardware feature. For example, loop invariant load removal cannot be performed if there is one store whose address may be the same as the load. With preload register update, the invariant load can be removed from the loop without compromising correctness. We are currently investigating new optimization algorithms to take full advantage of preload register update.

Acknowledgements

The authors would like to acknowledge Bob Rau and Mike Schlansker at HP Labs, along with all members of the IMPACT research group for their comments and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Co. Ltd., Hewlett-Packard, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] 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 *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.

[2] M. J. Wolfe, *Optimizing Compilers for Supercomputers.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1982.

[3] J. R. Allen, *Dependence Analysis for Subscripted Variables and Its Application to Program Transformation.* PhD thesis, Department of Mathematical Science, Rice University, 1983.

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

[5] Z. Li, P. Yew, and C. Zhu, "Data dependence analysis on multi-dimensional array references," in *Proc. Int'l Conf. on Supercomputing*, (Crete, Greece), June 1989.

[6] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *Proc. 1991 SIGPLAN Symp. Compiler Construction*, pp. 15–29, June 1991.

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

[8] 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.

[9] 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 *Proc. 24st Ann. Workshop on Microprogramming and Microarchitectures*, (Albuquerque, NM.), Nov. 1991.

[10] 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.

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

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

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

[14] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, Dec. 1991.

[15] 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.

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

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

[18] 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 *Proc. Second Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Palo Alto, CA), pp. 180–192, Oct. 1987.