An Efficient Architecture for Loop Based Data Preloading

William Y. Chen Roger A. Bringmann Scott A. Mahlke Richard E. Hank James E. Sicolo

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

Abstract - - Cache prefetching with the assistance of an optimizing compiler is an effective means of reducing the penalty of long memory access time beyond the primary cache. However, cache prefetching can cause cache pollution and its benefit can be unpredictable. A new architectural support for preloading, the preload buffer, is proposed in this paper. Unlike previously proposed methods of non-binding cache loads, the preload is a binding access to the memory system. The preload buffer is simple in design and predictable in performance. With simple interleaving, accesses to the preload buffer are independent of the access pattern and processor issue rate, and are therefore free of bank conflicts. With trace driven simulation, it is shown that the performance from preloading hides memory latency better than no prefetching and cache prefetching. In addition, both the bus traffic rate and the miss rate are reduced.

1 Introduction

Cost versus performance tradeoffs have resulted in system designs with fast processors, and diverse speeds of memory sub-systems. The disparity between the speeds of the memory sub-systems 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 paper focuses on methods of compiler-assisted data prefetching to reduce this latency.

Previous work has shown compiler-assisted prefetching to be an effective means of reducing the penalty of long memory access time beyond the primary cache [1] [2] [3] [4] [5]. Loop restructuring [6] [7] can be used to improve local memory performance by grouping references to the same memory location together to improve the utilization of the first level cache. Data prefetching and loop restructuring can be combined effectively to improve the overall system performance. However, there are problems associated with compiler-assisted prefetching. It is the goal of this paper to introduce these problems and to provide a new architecture to resolve them. The concept of *preloading* is introduced to complement cache prefetching. Unlike cache prefetching, preloads are binding accesses to the memory system. Preloading and its architectural support are described in detail later.

1.1 Cache Prefetch

The main objective of data prefetching is to decrease the overall execution time of a given application. Advanced hardware based prefetching methods have been proposed to let the hardware detect access patterns and decide when to perform a cache load [8] [9]. Conventional means of compiler-assisted data prefetching require the compiler to generate a non-binding memory load to the cache block. This special prefetch instruction informs the memory sub-system that a piece of data may be used in the near future. According to this hint, the data is fetched into the cache if not already present. For prefetching to be effective, at least the first level cache must be non-blocking, allowing prefetch accesses to overlap with other memory accesses and computations. Also, non-trapping hardware is needed to remove constraints which hinder moving prefetch instructions above conditionals.

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.

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 [3]. 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 is referenced, its cache line 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 FIFO queue, 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 pollution within the prefetch buffer and causes additional unnecessary memory traffic.

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.

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. Register preloading has shown to be tolerant of the memory latency, however, at a large increase in register usage [10]. This increase in number of registers requires non-conventional 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.

1.4 Prefetch for Superscalar Processors

To sustain its performance, a superscalar processor must be able to issue multiple memory loads per cycle [11]. 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 load can be executed per cycle, increasing the number of instructions bypassed leads to more memory loads 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.

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. To maintain 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 must 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 paper presents a simple architectural design based on the above criteria. This scheme 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 3 the new prefetch architecture and compiler support are described in detail. Section 5 presents the simulation result of the new prefetch architecture. Concluding remarks are given in Section 6.

2 Cache Prefetch

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 4.

All of the results reported are obtained using execution driven simulation based on actual compiled code. Within the context of this paper, 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 1 are used for our simulation. The 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 block size 32 bytes. These caches are non-blocking with no limit on outstanding requests.

Table 1: Instruction latency.

INT function	*latency*	*FP function*	*latency*
ALU	1	ALU	3
barrel shifter	1	conversion	3
multiply	3	multiply	3
divide	10	divide	10
load	2	load	2
store	1	store	1

Table 2: 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%

Benchmarks with a significant level of cache misses should be used to evaluate prefetch schemes, since cache prefetch is only needed when the miss ratio is high. The miss ratios for five benchmarks are shown in Table 2. Two of the benchmarks, *doduc* and fpppp, exhibit relatively low miss ratios. By increasing the cache size, the misses have almost completely disappeared. Therefore, these benchmarks will not be discussed further. High miss ratios, however, are observed for *matrix300*, *nasa7*, and *tomcatv*, even with a 256K byte cache. Since increasing the cache size does not sufficiently reduce the miss ratios, cache prefetching is considered as an alternative solution to improve 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 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 into the cache from the prefetch buffer upon use. The black bar, closely represents the ideal prefetch case and serves as a good basis for improvement over the other two results. [1] The black bar has non-zero 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 the result for the 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 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 has 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 the problem of block replacement and pollution to be 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.

Miss ratio alone does not give a complete picture of the

[1] Although the miss ratio is low, the penalty for bus traffic is overly high since every prefetch is required to go off-chip. The bus traffic associated with an ideal prefetch buffer is therefore omitted from the paper.

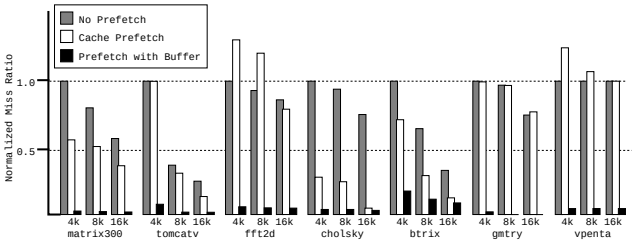

Figure 1: Comparison of normalized miss ratios.

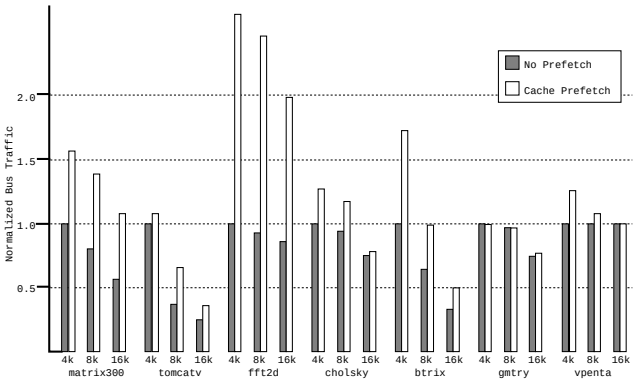

Figure 2: Comparison of normalized bus traffic.

effects of cache prefetching. Bus traffic must also be evaluated with respect to system performance. Figure 2 plots the normalized bus traffic for the no prefetch and prefetch cases. It is not surprising that the benchmarks with increased miss ratio due to prefetching exhibitted 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 decrease proportionally.

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 5 using a more detailed system simulation.

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 upon use.
- The preload buffer is a separate module from the data cache. No communication between the data cache and

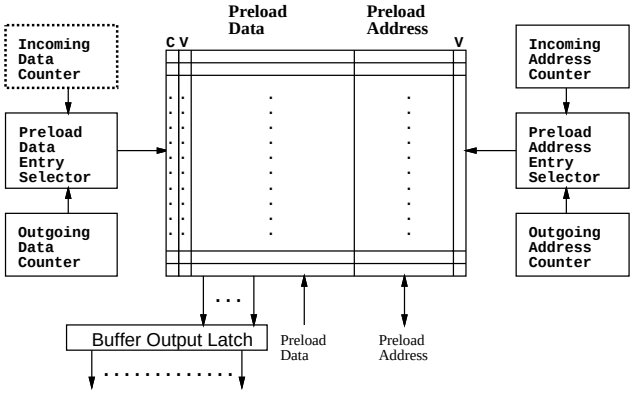

Figure 3: Hardware support for the preload buffer.

the preload buffer is necessary.

- The preload buffer is employed to complement cache prefetch.
- No comparison of address tag 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 all data within the preload buffer 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.

3.1 Hardware Organization

The hardware support for preload buffer is shown in Figure 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 a pointer to some entry within 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) bit marks all the valid entries. The (C) bit indicates the possibility of a stale data within the preload buffer. The number of entries within the preload buffer is an architectural parameter that must be made known to the compiler in order 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 excercises 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, memory hazards and conditionals are dealt with the *prestore*, *storeb* and *skip* instructions.

3.2 Basic Concept

The preload buffer views the memory loads within a single loop iteration with sequential ordering. 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 insturctions, the *init* instruction is executed. This resets 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 same address register is set to 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 only permit the memory system to return in-order preload data. We assume the existence of an incoming data counter for our simulation results. The requested preload address entry is set to invalid when the request is sent.

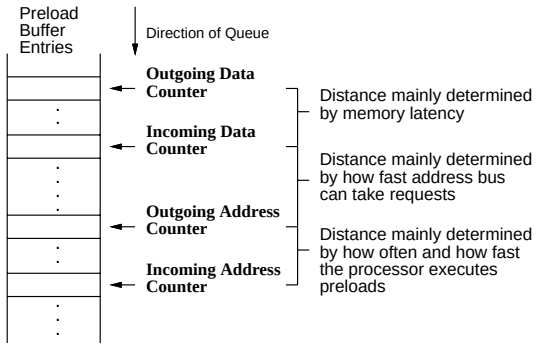

Figure 4: An ordering for the address and data counters.

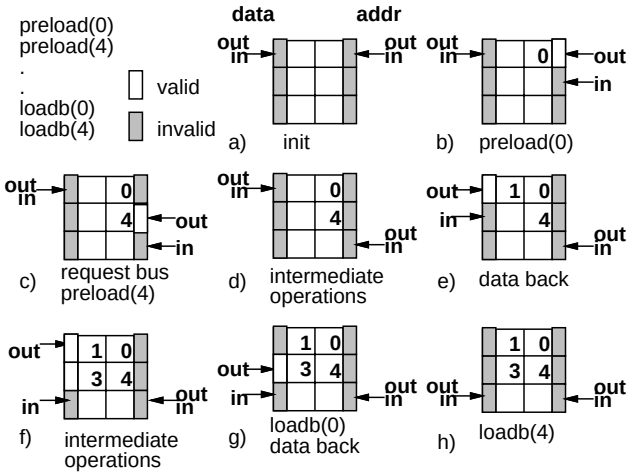

Figure 5: An example operation of the preload buffer.

The preload data entry is set to valid when the requested data is received. Due to the lag time in 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 is 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 3.3 and 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 the preload address registers. This prevents the preload

without prestore	with prestore
pld(60)	pst(60)
pld(20)	pst(20)
	pld(60)
	pld(20)
st(60)	stb(60)
pld(60)	pld(60)
st(20)	stb(20)
ldb(60)	ldb(60)
ldb(20)	ldb(20)
pld(20)	pld(20)

Figure 6: Example coherence problem and solution.

buffer from sending residual preload requests resulting from the epilogue of the preloaded loop.

Due to system specifications 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 4. The operation of the preload buffer is shown with a simple example given in Figure 5. The states of the preload buffer are shown from a to h for the code segment given.

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. The store addresses of 20 and 60 matches the loadb addresses. Therefore, the previous preloads for these addresses obtain the stale value since an intermediate store has modified the same address.

To ensure correct program execution, we cannot preload a load if the preload needs 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 7) are added to the 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

[2] We define short dependence distance as a number of processor cycles that is less than the memory latency, which prevents effective insertion of latency hiding preloads.

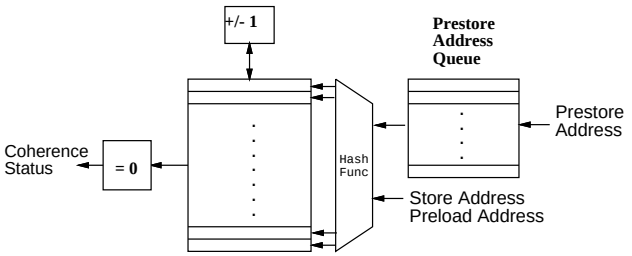

Figure 7: Design of a prestore log to maintain coherence.

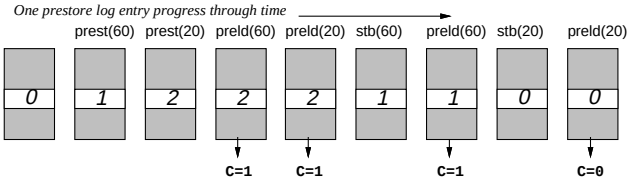

Figure 8: Example execution using the prestore log.

correctness of the data preloaded. Prestore does not make a memory request; it only affects 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 which hash 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 the entry is unavailable due to coherence problem, the load request is routed to the data cache.

Figures 6 and 8 are used to clarify the concept. Addresses 60 and 20 hash into the same prestore log entry. When prest(60) and prest(20) obtain the bus, no memory request is made. The address 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 note 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 eventhough the second preload from address 60 can obtain the correct memory content, the preload is aborted due to mapping conflicts. The prestore log can be made less conservative by increasing the number of log entries to decrease the chance of mapping conflicts. Later preloads and loadbs operate normally.

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

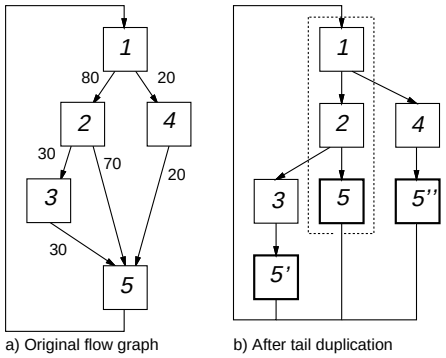

Figure 9: Example flow graph with conditionals.

3.4 Dealing with Conditionals

All load instructions and their preload data entry location 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 method.

Consider the flow graph in Figure 9a. 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 non-constant 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 the 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 preforming tail duplication, a single path of control is extracted, and the resulting control flow is shown in Figure 9b. 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. This involves insertion of preloads and 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 individual exits of the main path into individual side paths and leave all memory accesses in the side paths as cache loads. In terms of Figure 9, 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". The program will execute correctly at this point.

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

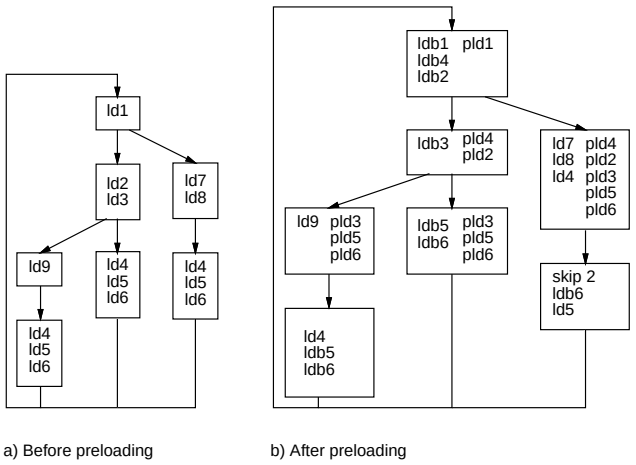

Figure 10: Example of preloading for conditionals.

value. Figure 10 shows an example before and after preloading for conditionals.

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.

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 just to save the outgoing data counter and the incoming address counter (Figure 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 the entries from the outgoing data counter up to the outoging 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 correct data is fetched for later loadb accesses.

4 Algorithm

```
for (all loads in loop)
        mark load as preload
trace selection and tail duplication
if (main path is not a self loop) exit
loop unrolling
for (marked loads in main path)
        mark all address producers
        if (address calc variant in loop)
                mark load as prefetch
                stride info = not ok
        if (address producer a load)
                mark load for second pass
        if (backward dependence exist)
                if (definite dependence)
                        mark load for prefetch
                if (maybe dependence)
                        mark store for prestore
                        mark store as address producer
code schedule main path
n = number of prefetch iterations
do (1 to n for main path)
        if (stride info ok)
                insert prefetch or preload in preheader
        else insert address producers in preheader
                insert prefetch or preload in preheader
                rename registers
for (main path)
        if (stride info ok)
                insert prefetch or preload in loop
        else insert address producers in loop
                insert prefetch or preload in loop
                rename registers
for (all side paths)
        code schedule path
        set = address producers before preload in main path
        set = set - address producers executed before exit
        insert producers of set in path
        insert preloads
        unmark loads that are speculative in main path
        for (marked loads in path)
                compare with preloads in main path
                if (load is in sequential preload ordering)
                        change load as preload
                else insert skip
                        unmark intermediate out of sequence loads
perform second pass
```

Figure 11: Prefetch/preload algorithm for loops.

In order 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 [11]. 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 the cache profiling done in this paper. By using the miss rate information, unnecessary prefetches to data that tend to be in the cache can be eliminated. References with non-zero miss rate will be prefetched for this paper, 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 producers of 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 by the compiler to determine the value of an invariant access stride can be used. 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. The address producing loads are not marked in the first pass.

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.

A pseudo code algorithm is presented in Figure 11 to combine all the concepts.

5 Experimental Evaluation

Experiments are conducted to evaluate the performance of preloading. The performance 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.

Table 3: Maximum number of preload buffer entries used.

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

5.1 Simulation Architecture

The base architecture is an in-order superscalar processor capable of fetching four instructions per cycle. Non-trapping 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 presented in Table 1. The primary cache size varies according to the simulation. In all cases, the primary cache is assumed to be non-blocking with an infinite sized miss queue. The second level cache is fixed at 1M byte 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.

5.2 Miss Ratio and Bus Traffic

During compilation to support preloading, the maximum number of preload buffer entries used per benchmark was recorded. Table 3 lists the benchmarks and their preload requirements. Gmtry requires the greatest number of entries. A maximum of 64 buffer entries were used for all later simulation results. To allow for comparison, 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 against 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 for no preload, 4K, 8K, and 16K byte caches. In all experiments, a preload data element that has not been validated before its associated load occurs is counted as a miss. Normalized miss ratio for preloading is shown in Table 4. When no preloading is performed, an increase in the

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

Table 4: Comparison of normalized miss ratio.

Bench	*No Prefetch*			*With Preload*		
	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

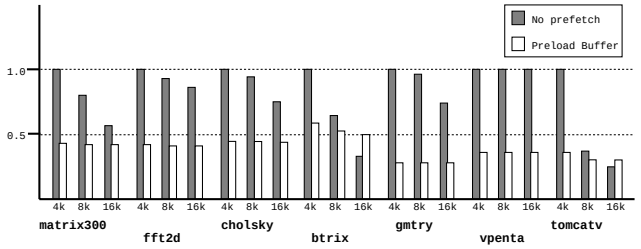

Figure 12: Comparison of normalized bus traffic.

cache size does not significantly reduce the number of misses for several of the benchmarks. By performing preloading, a significant percentage of misses are eliminated for all benchmarks.

Next, the bus traffic was evaluated. The normalized bus traffic ratio comparing no preload versus preload is shown in Figure 12. Surprisingly, preloading reduces the bus traffic for all but two cases. Since the miss ratio is always reduced, a decrease in bus traffic guarantees an improvement in system performance. Detailed processor simulation is required to show the extent that preload improves the overall system performance. Also, in the case of *btrix* and *tomcatv*, we need to find out how the increase in bus traffic affects the performance of preloading.

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.

The 8K and 16K cache results are shown in Figure 13 and 14 respectively. It is not surprising that memory overhead accounts for a large portion of the execution time in an issue 4 superscalar processor. Also, due to the increase in bus traffic, cache prefetching improves the performance of

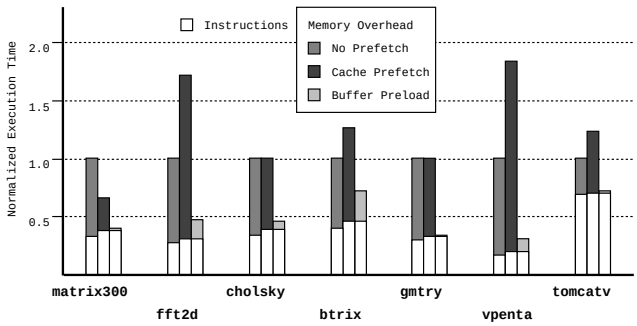

Figure 13: Comparison of normalized execution time for 8K cache.

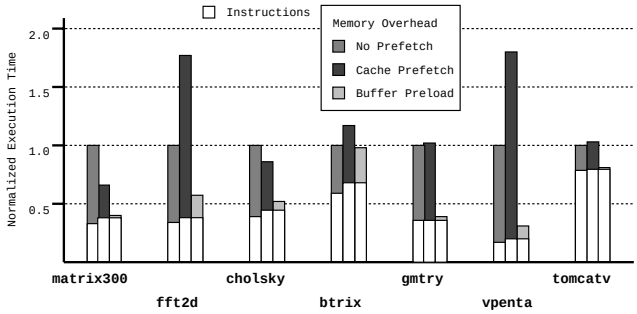

Figure 14: Comparison of normalized execution time for 16K cache.

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

5.4 Effects of Cache Block Size

Up to this point all 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. Thus, any decrease in cache performance is due to mapping conflicts between references. A small block size is beneficial in reducing cache polution for long stride accesses. However, a small block size creates extra bus traffic which degrades system performance for short stride accesses.

In Figure 15, the results for an 8K cache are shown. All results are normalized against the result of no prefech using an 8K cache with a 32 byte block. A noticeable performance shift can be seen for no prefetch and cache prefetch. *Matrix300* performance significantly dropped due to the benefit of long cache block fetches. The elimination of unnecessary cache block fetches greatly improved the performance of the other benchmarks. The benefits of cache prefetch, however, are still unpredictable. The unpredictability of the cache prefetch performance is mainly due to conflicts between the present working set and the future working set. Also the

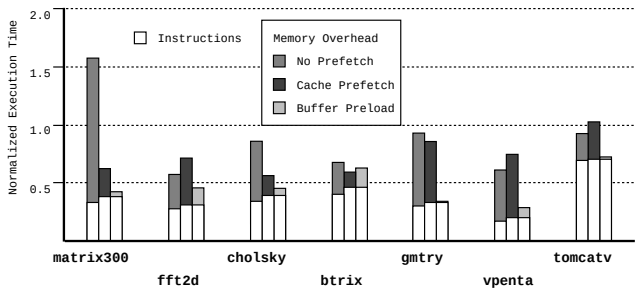

Figure 15: Comparison of normalized execution time for 8K cache with 8 byte blocks.

mapping conflicts of these references also contribute greatly into the utilization of the cache. The effective cache size may be smaller than its physical size dependending on how the structures are mapped onto the cache. These conflicts may remove useful data from the cache, thus increase the bus traffic to reobtain these data from off-cache storages.

It is interesting to note that the extra bus traffic finally caught up with *btrix*, and cache prefetch now performs slightly better than the preload buffer. In comparison to Figure 13, the main observation is the performance of the preload buffer stays relatively constant. This predictability can contribute greatly in determining when to perform preloading.

6 Conclusion

This paper proposes an architectural support, the preload buffer, to cooperate with the compiler in tolerating long memory access latencies. Unlike previously proposed methods of non-binding cache loads, a preload is a binding access to the memory system. The prefetch buffer requires associative searches and 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 between data arrays. The preload buffer attempts to eliminate 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 problem 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.

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.

Acknowledgements

The authors would like to thank all members of the IMPACT research group for their comments and suggestions. This research has been supported by JSEP under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, NSF under Grant MIP-8809478, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Co. Ltd., Hewlett-Packard, NASA under Contract NASA NAG 1-613 in cooperation with ICLASS.

References

[1] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessor with memory hierarchies," in *Proc. Int'l Conf. on Supercomputing*, (Amsterdam, The Netherlands), pp. 354–368, June 1990.

[2] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proc. Fourth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 40–52, Apr. 1991.

[3] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 43–53, May 1991.

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

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

[6] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proc. of the 4th SIAM Conference*, 1989.

[7] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel and Distributed Computing*, vol. 5, pp. 344–358, 1988.

[8] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 54–63, June 1991.

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

[10] W. Y Chen, S. A. Mahlke, and W. W. Hwu, "Tolerating first level memory access latency in high-performance systems," in *Proc. 21th Int'l Conf. on Parallel Processing*, Aug. 1992.

[11] 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.