Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching

William Y. Chen *Scott A. Mahlke* *Pohua P. Chang* *Wen-mei W. Hwu*

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

Abstract

The performance of superscalar processors is more sensitive to the memory system delay than their single-issue predecessors. This paper examines alternative data access microarchitectures that effectively support compiler-assisted data prefetching in superscalar processors. In particular, a prefetch buffer is shown to be more effective than increasing the cache dimension in solving the cache pollution problem. All in all, we show that a small data cache with compiler-assisted data prefetching can achieve a performance level close to that of an ideal cache.

1 Introduction

Superscalar processors can potentially deliver more than five times speedup over conventional single-issue processors [1]. With the total execution cycle count dramatically reduced, each cycle becomes more significant to the overall performance. Because each data cache miss can introduce many extra execution cycles, a superscalar processor can easily lose the majority of its performance to the memory hierarchy.

Out-of-order execution can partially hide the miss penalty [2]. However, its performance is limited by the instruction window size. A sophisticated compiler can increase the distance between a memory load and the usage of its result via code optimization and scheduling. The problem is that the movement of memory operations is limited by the effectiveness of memory disambiguation. In general, neither method can effectively eliminate the performance degradation due to data cache misses.

In a compiler-assisted data prefetching scheme, an intelligent compiler inserts data memory prefetch operations many cycles before their corresponding memory accesses. Each data memory prefetch operation performs a memory load to a data cache block or a prefetch buffer entry instead of a register. By making intelligent decisions as where to insert a prefetch operation, the appropriate data will be available in the cache or prefetch buffer when the corresponding memory operation is executed. Compiler-assisted data prefetching does not depend on accurate memory disambiguation because prefetching changes the cache states rather than the register values. Therefore, compiler-assisted prefetching can be much more aggressive in hiding cache miss penalty than code scheduling.

Researchers have concentrated on compiler-assisted data prefetching in the loop domain for single-issue processors. Callahan, Kennedy, and Porterfield have examined the effect of inserting specific prefetch operations for subscripted variables one loop iteration ahead at the source code level [3]. Klaiber and Levy have gone further by performing loop-based data prefetching at the assembly code level. A fix-sized prefetch buffer is used to hold the prefetched data [4].

In this paper, we focus on the design of data access microarchitectures to support data prefetching. More specifically, we address the problem of cache pollution due to data prefetching. Cache pollution can translate into cache misses, which in turn defeats the purpose of prefetching. The ability to minimize the degrading effect of cache pollution by increasing the cache dimension (cache size and/or set associativity) is compared against that of a separate prefetch buffer. Hardware issues concerning both approaches are discussed. The ability of different issue rates to hide the data prefetching overhead will be judged in terms of the net processor speedup.

2 Compiler Issues

The main concept behind compiler-assisted data prefetching is to utilize the compiler to insert prefetch operations in advance so that the needed datum will be available in the data cache when the actual memory operation is executed. By performing data prefetching at the assembly level instead of at the source code level, estimating the execution-time of program segments can be more accurate. If the memory latency is L cycles, then a prefetch operation must be inserted at least L cycles ahead in the execution stream. For higher operation issue rates, more operations have to be bypassed in order to insert the prefetch operation. Address calculations may have to be duplicated and placed before the prefetch operation.

[0]

```
L: r1 <- mem(r2)
   r3 <- r3 + 1
   r2 <- r2 + 4
   if (r3 != r1) goto L
```

Figure 1: An example with no data prefetch.

```
memory latency = 2          memory latency = 8

   prefetch(r2)                prefetch(r2)
   ......                      ......
L: r1 <- mem(r2)               r4 <- r2 + 4
   r2 <- r2 + 4                prefetch(r4)
   prefetch(r2)             L: r4 <- r4 + 4
   r3 <- r3 + 1                prefetch(r4)
   if (r3!=r1) goto L          r1 <- mem(r2)
                               r2 <- r2 + 4
                               r3 <- r3 + 1
                               if (r3!=r1) goto L
```

Figure 2: Example code segment with data prefetching.

Destination registers of the duplicated operations are renamed to avoid conflict with the original code.

If the address calculations can all be duplicated, then there are no difficulties involved in inserting the prefetch operation for that particular memory operation. We can illustrate this by an example where all operations are assumed to require one cycle to execute. Figure 1 presents a loop with one memory load operation. With different memory latencies, the compiler will produce different data prefetching code as shown in Figure 2. For higher memory latencies, the register usage increases due to the duplicated address calculation. Since many of the operations are independent, they can be executed concurrently by high issue-rate processors.

Since loops do not dominate the execution time of many scalar programs, we consider data prefetching for both loop and non-loop segments. For both types of code segments, each prefetch operation and the associated address calculations must be placed in all program paths to satisfy the L cycles requirement. To insert a prefetch operation for a memory operation from a code segment, care has to be taken as not to consider a loop if this loop does not contain the code segment. By inserting a prefetch operation into a wrong loop, not only do we waste memory bandwidth prefetching useless data and cause memory pollution, but we may also increase the actual execution cycles. For memory operations within a loop segment, we have to consider adding the prefetch operation within the loop and outside the loop. Prefetch operations outside the loop correspond to the first few loop iterations. Prefetch operations within the loop are for the later loop iterations.

Sometimes there are constraints that will not allow the compiler to insert the prefetch operation as far ahead as desired. This is because the address of many memory operations are dependent on other memory load operations. We refer to the memory operation that uses the loaded address (by an *address load*) as a *dependent memory operation*. Figure 3 shows the commonly used *getc* function in C to illustrate this point. Since the address contained in register $r1$ is the result of another memory load, the value in $r1$ will not be available until the first memory load completes its execution. Therefore, the earliest point to insert the prefetch operation is right after the first memory load operation. The resulting code with data prefetching is shown in Figure 3.

```
r1 <- mem(r2 +4)          prefetch(r2 + 4)
r3 <- r1 + 1              ......
mem(r2 + 4) <- r3         r1 <- mem(r2 + 4)
r4 <- mem(r1)             prefetch(r1)
                          r3 <- r1 + 1
                          mem(r2 + 4) <- r3
                          r4 <- mem(r1)
```

Figure 3: Example code segment with a dependent memory operation.

Due to address loads, the cycles between the prefetch operation and the corresponding memory operation can be smaller than L. In this case, prefetching cannot completely hide the cache miss penalty. But since the data cache refill has been initiated by a prefetch operation, the penalty cycles are reduced.

3 Hardware Issues

Several issues and tradeoffs must be considered when dealing with the hardware architecture. First, addresses for the prefetch operations may cause exceptions, and non-trapping hardware must be available to ensure continuing execution of the program. A prefetch operation which induces a fault will be simply discarded. Colwell *et al.* have already implemented an architecture with non-trapping supports [5]. Second, prefetch operations must be non-blocking so the hardware does not stall by a miss caused by a prefetch operation. The design of a non-blocking cache for superscalar processors has been considered recently by Sohi and Franklin [7]. Multiple outstanding requests can be handled by the method discussed by Kroft [6]. Third, the problem concerning where to place the prefetched data must be examined. In this paper, we will concentrate on the study of the third issue. For the remaining of this section, considerations for the design of a prefetch data cache and the design of a separate prefetch buffer are discussed.

Prefetched data can be directly placed into the data cache. Because of different execution paths, the prefetched data are not guaranteed to be useful in the near future, while the replaced block may be accessed during this time interval. We can reduce this effect of cache pollution by either increasing the cache size, the set associativity, or both. Increasing the cache size and/or the set associativity of the cache can reduce the chance of replacing a useful block. Still, we are introducing many useless entries into the data cache, and the pollution problem remains, although at a reduced level.

The data bandwidth is a problem for the prefetch cache. If we wish to maintain the same cache bandwidth, then the CPU will always have priority over the prefetch activities in cache access. The prefetched data will be placed into the cache after the cache has serviced the CPU. This

	general		
program	*issue 2*	*issue 4*	*issue 8*
eqntott	30.3%	38.8%	47.7%
espresso	37.5%	39.1%	39.2%
tbl	16.8%	19.0%	19.8%
xlisp	41.4%	44.7%	45.0%
yacc	18.9%	19.8%	21.3%

Table 1: Percentage of dependent memory operations.

may decrease the effectiveness of data prefetching. On the other hand, if we wish the data cache to service both the memory operations and the prefetch operations simultaneously, the bandwidth of the cache will have to increase to keep up with the service rate. The datapath to the cache, therefore, must be duplicated to perform simultaneous reads and writes to/from the cache.

Prefetched data can be placed into a separate prefetch buffer with block size the same as that of the data cache [4]. When a memory operation is executed, both the data cache and the prefetch buffer are checked. The CPU will retrieve the proper entry from the data cache first before the prefetch buffer. If the entry is not in the cache and is in the prefetch buffer, the data is forwarded to both the CPU and the data cache. If a miss occurs in both the cache and the prefetch buffer, a cache miss is assumed and is handled normally. If a load hits in both the cache and the prefetch buffer, the datum from the data cache and the prefetch buffer is multiplexed to give priority to the data cache [1]. For all stores that hit in the data cache, the corresponding buffer entries are invalidated. Otherwise, the prefetched block is transferred to the data cache before the invalidation of the buffer entry. The data cache does not have to be non-blocking, but the prefetch buffer must be able to handle multiple outstanding memory requests. The bandwidth of the data cache remains the same. An extra communication channel is needed between the data cache and the prefetch buffer for data transfer. Since the prefetch buffer is concurrently accessed along with the data cache by the CPU, the prefetch buffer suffers the same bandwidth problem as the prefetch cache. The difference is that the prefetch buffer has no dirty-block state, and thus it does not have to worry about the simultaneous stores from both the memory and prefetch operations. Therefore, the prefetch buffer states are simpler than those of the prefetch cache.

With a prefetch buffer, we can guarantee that all data entering the cache will be used at least once. In comparison, for prefetching into the data cache, the cache size and/or associativity could be increased to reduce the degrading effect of pollution, however, useless data cannot be prevented from entering the cache.

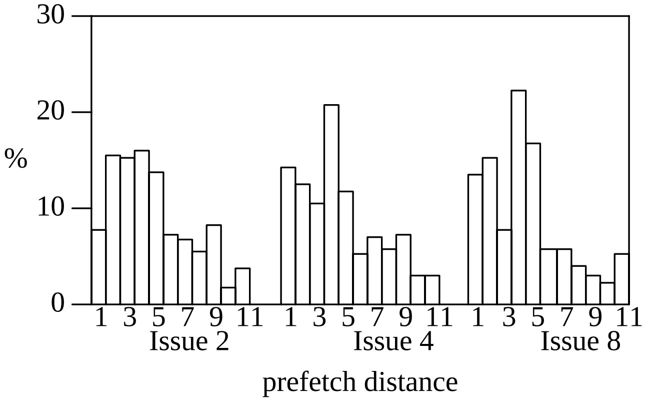

Figure 4: Eqntott and espresso prefetch distance.

4 Simulation Environment

Table 1 lists the 5 benchmark programs used in this paper. The measurement results are generated by trace driven simulation. The traces consists of the IMPACT assembly instructions, LCODE, which is a superset to the MIPS R2000 assembly language. The trace analyzer consumes the entire instruction trace while the benchmark program executes. The architecture used assumes in-order execution with register inter-locking and renaming, and uniform function units for all issue rates. The issue rate is the maximum number of operations that can be dispatched per cycle. The basic processing element has deterministic operation latencies. All integer operations have a 1 cycle latency with the exception of multiply (3 cycles) and divide (25 cycles). The memory load latency is 2 cycles. Finally, all floating point operations have a 3 cycle latency with the exception of multiply (4 cycles) and divide (25 cycles). Since prefetching may cause exceptions, non-trapping hardware is assumed. The trace analyzer can simulate the effect of prefetching data either into the data cache or into the prefetch buffer. All caches have 32-byte blocks, and the cache repair time is assumed to be 10 cycles. The prefetch buffer is a fully-associative FIFO queue with the same block size as the data cache.

For conventional commercial processors, the restricted code percolation model is used. With a complete set of non-trapping operations, general code percolation can move independent loads, divides, and floating point operations above branches to further increase the processor performance (restricted code percolation does not allow operations which may cause traps to be moved above branch operations) [8]. We will compare the prefetching results between restricted code percolation model and general code percolation model. A trap caused by a prefetch operation is handled the same as a trap caused by a memory operation within the general code percolation model.

5 Address Loads

Since address loads affect the performance of data prefetching by limiting the prefetch distance, we wish to study the characteristics of the address loads within the benchmark programs using the general code percolation model. Table 1 lists the dynamic percentage of address loads that are detected within the prefetching range (which is 10 cycles in this case). The remaining memory

[1] Since the data cache has a much lower set-associativity than the prefetch buffer, the cache can access the data faster. Assuming higher percentage of hits than misses, data access time is usually that of the faster cache instead that of the slower prefetch buffer.

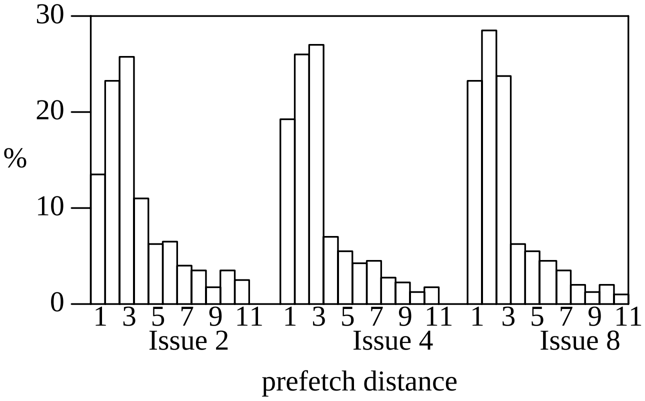

Figure 5: Tbl, yacc, and xlisp prefetch distance.

operations (e.g. 1 - x% of the memory operations from Table 1) have no constraints in upward code motion, and their prefetch distance is always the maximum allowable.

Figures 4 and 5 show the actual distribution of the distance between address loads and dependent memory operations for different issue rates. This is the maximum prefetch distance for the dependent memory operations since we do not allow movement beyond a address load. For our scheduling model, we have already scheduled the address loads as much as possible. The basic distribution can be separated into two categories. The average result for *eqntott* and *espresso* is shown in Figure 4. The average result for *tbl*, *xlisp*, and *yacc* is similarly shown in Figure 5. The main disparity between the two Figures is that for *eqntott* and *espresso*, the maximum prefetch distances tend to be evenly distributed across the entire spectrum. For the other three benchmarks, the maximum prefetch distance is concentrated at distances of 3 or less.

The effectiveness of the prefetching approach used in this paper is limited by the constraints of the address loads. For example, we can see that *xlisp* has a high percentge of memory operations that require the use of an address resulting from another memory load (Table 1). In conjunction with the results from Figure 5 (where prefetch distances are small), we can expect prefetching to be less effective for *xlisp* as opposed to a lesser constrained program such as *espresso*. This behavior is indeed observed for *xlisp*.

6 Hardware Tradeoffs

Since data prefetching has the potential to increase cache pollution, we wish to minimize the degrading effect of cache pollution through different data access considerations. Figures 6 and 7 show the degree of effectiveness of increasing cache size and/or set associativity versus using a prefetch buffer. The use of several configurations of prefetch buffer is evaluated in comparison with the case of a perfect cache (indicated by *per* in each figure), and the case of a 1K direct mapped data cache with no prefetching (indicated by *1dX* in each figure). The first number in each of the configuration represents the cache size in 2^{10} bytes. The second letter represents the cache associativity (e.g. *d* is for direct mapped, and 2 is for 2-way associative). The third letter represents the prefetch buffer size in 2^x entries (e.g. *n* is for prefetch into the cache, and a 3 is for a 2^3 entry buffer).

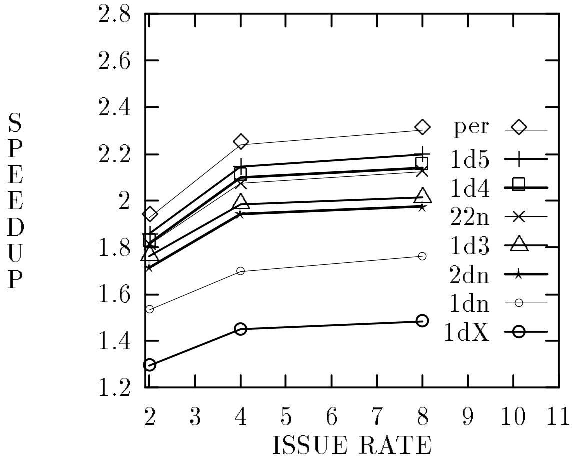

Figure 6: Prefetch results (restricted).

We plot speedup versus the issue rates of 2, 4, and 8. Each data point represents the average of the 5 benchmarks given in Table 1. The base architecture for all speedup calculations has an issue rate of 1, a 1K direct mapped cache, no prefetching, and uses the restricted code percolation model. For all results of data prefetching with a prefetch buffer, a 1K direct mapped cache is used. Since close to ideal speedup can be achieved by utilizing data prefetching and a 32 entry prefetch buffer on top of the base architecture, the use of a 1K direct mapped cache is justified for all our results.

From Figures 6 and 7, it can be seen that superscalar processors can indeed lose a majority of their performance to data cache misses. However, data prefetching can be used to effectively alleviate most of the performance loss. It is shown that most configurations with data prefetching perform better than the configurations without data prefetching. There is no noticeable performance degradation due to the extra overhead incurred by the prefetch operations and the address calculations for the issue rates shown.

Prefetching improves the performance more than increased issue rate. Increasing the issue rate requires the duplication of data path and function units. Prefetching requires the use of non-trapping and non-blocking hardware. In comparison, performing prefetching achieves a higher performance with lower hardware cost. For the small sized caches, we can conclude that it makes more sense to perform data prefetching before an increase in the issue rate.

It is clearly indicated that a 1K direct mapped cache with a 32 entry prefetch buffer performs best for all benchmarks. For a direct mapped cache, increasing the cache size is not as useful as adding an 8 entry prefetch buffer. Increasing the associativity of the cache decreases data conflicts within the cache and removes some effects of

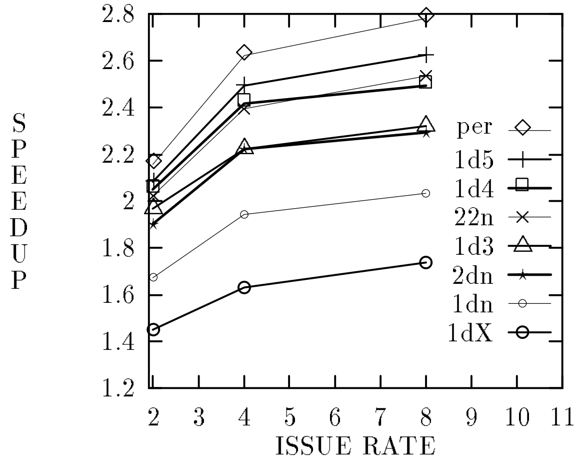

Figure 7: Prefetch results (general).

cache pollution. For restricted code percolation model, *22n, 1d4, and 1d5* exhibit similar behavior, but the curve for *1d5* noticeably out performs *22n and 1d4* under the general code percolation model.

The general percolated code gives noticeable performance gain over the code with restricted percolation. However, without data prefetching, the processor performance for the general code percolation model suffers a greater loss than restricted code percolation model. From this, we conclude that there is a greater need to perform data prefetching for general code percolation model.

7 Conclusions

Superscalar processors can lose the majority of their performance to data cache misses, but we have shown that compiler-assisted data prefetching significantly reduces the penalties associated with data cache misses. We achieve higher performance by performing compiler-assisted data prefetching than by increasing the issue rate. From the data shown, we see a very strong need to perform compiler-assisted data prefetching for superscalar processors to achieve its potential performance limit.

Over all the benchmark programs examined, the addition of a prefetch buffer is preferred over an increase in cache dimensions. For the benchmark programs examined, a 32 entry prefetch buffer seems to be enough to capture most of the prefetched data. There is no significant performance degradation due to the extra overhead incurred by the prefetch operations and the address calculations for the issue rates of 2, 4, and 8.

General code percolation significantly increases the processor performance, but the degrading effect of data cache misses is also greater than restricted code percolation model. Since the non-trapping hardware is already available for the prefetch operations, at a modest increase of the instruction set, we can increase the processor performance by utilizing both the compiler-assisted data prefetching and the general code percolation model.

Finally, in this paper we have focused on the effects of compiler-assisted data prefetching on a set of scalar programs. Currently we are using programs with larger cache requirements to evaluate the effects of prefetching on larger data caches.

Acknowledgements

The authors would like to thank Tom Conte and all members of the IMPACT research group for their support, comments and suggestions. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsuhita Electric Corporation, 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] M. Butler, T. Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, “Single instruction stream parallelism is greater than two,” in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 276–286, June 1991.

[2] W. Y. Chen, “An optimizing compiler code generator: A platform for RISC performance analysis,” Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 1991.

[3] 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.

[4] 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.

[5] 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.

[6] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization,” in *Proc. 8th Ann. Int'l Symp. Computer Architecture*, (Minneapolis, MN), pp. 81–87, May 1981.

[7] G. S. Sohi and M. Franklin, “High-bandwidth data memory systems for superscalar processors,” in *Proc. Fourth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 53–62, Apr. 1991.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, “IMPACT: An architectural framework for multiple-instruction processors,” in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.