Grouped Prefetching: Maximizing Resource Utilization

Weston Harper, Justin Mintzer, Adrian Valenzuela
Rice University – ELEC525 Final Report

Abstract

Prefetching is a common method to prevent memory stalls, but it depends on the time sensitive return of prefetch requests relative to their consumption by an operation. Additionally, adding prefetches increases the contention on memory system buses which is counterproductive to ensuring the timely service of prefetches. By issuing prefetches as single grouped access through memory buses, bus contention can be reduced. However, this is improvement in bus contention over normal prefetching is only relevant for applications which place sufficient stress on the memory system to saturate the buses. This may be an increasingly common occurrence as processors continue to speed up faster then memory systems.

1 INTRODUCTION

As the response time of memory relative to processor speed continues to degrade, more focus is being placed on attempting to avoid paying the increasingly large memory stalls. One group of methods that has been developed to address this issue is the prefetching of instructions or data into memory before they are actually called for by an operation. However, the ability of a prefetch to convert a hit into a miss is dependent on whether the prefetch is serviced before the information is required for an operation.

There are many factors that affect how quickly a prefetch can be serviced, including the hit latency of structures in the memory hierarchy, memory bus speed, memory bus contention, and contention in the memory hierarchy with other memory requests that require servicing. The addition of many prefetches to a memory service stream can potentially have a negative impact on the ability of the memory system to service normal memory requests, as they are now forced to contend with the prefetches. The role of bus speed and contention in servicing prefetches would seem to indicate that better use of the memory hierarchy buses could improve the impact prefetches have on performance.

Motivation

First it will be useful to define some attributes that describe how well a prefetch system works. The *coverage factor* of a prefetcher is the fraction of cache misses that are changed to hits by prefetching. An *unnecessary* prefetch is one that does not need to be issued because the requested line is either already in the cache or a prefetch has already been issued to service that line. A *useless* prefetch is a prefetch that brings in a line that is evicted from the cache without ever being used. This is caused when something is prefetched and goes unused or when a prefetched line is brought in too early and evicted before it can be used. A *useful* prefetch is a prefetch that brings in a line that results in a cache hit before the line is replaced. These terms help describe how the *prefetch distance*, or the time between issuing a prefetch request and the use of a prefetch line, affect the *coverage factor*. A prefetch must be done early enough that it is ready to be used before a cache miss is registered, but not so early that it is evicted before being used and becoming a *useless* prefetch. Thus a critical factor in determining the effectiveness of a prefetcher is its *timeliness*. [1]

Assuming that it is more difficult to ensure that prefetch arrives early enough to hide a miss than it is to prevent a prefetch from arriving to early and becoming *useless*, we can examine the memory hierarchy buses as a factor in the *timeliness* of a prefetch. Under this assumption, it would be desirable to minimize the number of memory requests competing for bus access so that queuing delays are decreased. It would also be desirable to maximize the number of memory requests that can be sent to the next level of memory at a time so that the lower level memory structure can begin processing them sooner.

The question remains as to whether there exist multiple contending memory accesses during prefetching that would benefit from reduced contention. Figure 1 shows that with a next-n-line prefetcher. There is an increased number of references contending for access to main memory.

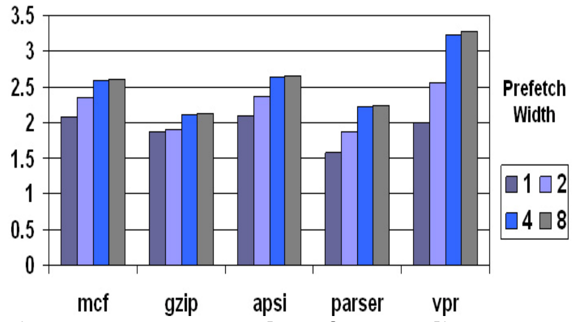


Figure 1: Average number of outstanding memory requests

Hypothesis

A major component of communication delay time for prefetch units is bus contention. By adding the ability for memory to handle group requests from prefetch units, the average bus contention delay per memory access can be significantly reduced.

In group prefetching, multiple prefetch requests would be calculated simultaneously and sent in a single packet to the bus instead of multiple bus access. Fortunately many existing prefetchers operate by generating a list of lines that the prefetch predicts will be needed in response to a single cache miss and thus provide a set of requests to group into a single bus access.

The rest of the paper is organized as follows. Section 2 describes the architecture of our proposed group prefetching method. Section 3 gives the experimental methodology we used, including the additions to simplescalar and the benchmarks we ran. Section 4 presents the results of the simulations we ran, and Section 5 concludes the paper.

2 ARCHITECTURE

Next-n-Line Prefetcher

A basic next-n-line prefetcher [2] will be used to generate prefetch requests. In the event of a miss in the cache, the logic that is attached to the prefetcher checks to see if the next N lines after the miss are in the attached cache and then issues a request for those lines which are missing.

Group Prefetch Communicator

In a normal next-n-line prefetcher this process would be done by issuing individual sequential requests across the bus to the lower level memory structure to service each prefetch. For our proposed architecture, an additional mechanism will be added to the prefetch unit. This prefetch communicator will collect the results of the check to see which next N lines actually require prefetching and store the results as a bit vector. Instead of sending each prefetch request sequentially, the prefetch communicator will send the base address of the first necessary prefetch, a bit flag indicating that a group prefetch is being made, and the sparse vector that indicates by offset from the base address the location of the other prefetches that require servicing by the lower level memory structure.

When the request reaches the lower levels of memory, the memory structure attempts to service as many of the prefetches as possible and returns the requested lines in a normal sequential fashion. If not all the requested lines are found then the sparse vector is updated to represent the requests that still require servicing. The base address and the update sparse vector are then forwarded by another prefetch communicator to the next lower level of memory. These stand-alone prefetch communicators are included with each level of memory between the level 1 caches that use prefetching and main memory.

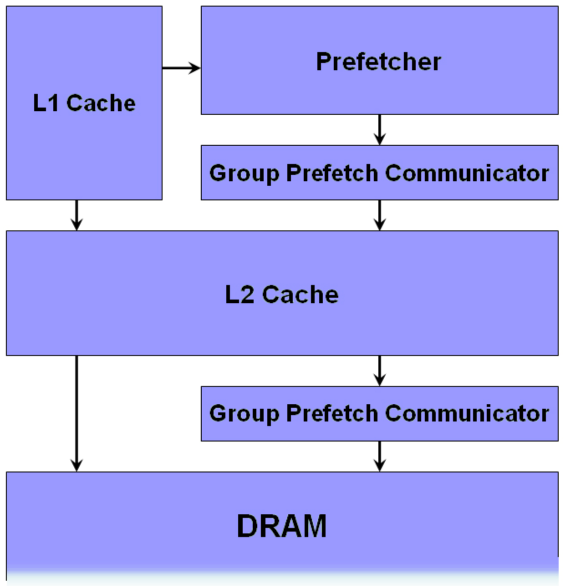


Figure 2: Memory hierarchy with modified prefetch communicator unit inserted

3 EXPERIMENTAL METHODOLOGY

Simulator

We choose to use Simple-Scalar 4.0 with Sim-Alpha configuration files to model the Alpha 21264 processor as our baseline architecture [3]. Simple-Scalar 4.0 was used over previous versions because it incorporates bus contention modeling which is critical to evaluating how our new prefetch communication model affects bus usage. To see the full system settings used, see the Appendix.

Additions to Sim-Alpha

Sim-Alpha's existing prefetch and code library was modified first to allow prefetching to be set for data level 1 cache in addition to just the instruction level 1 cache, and then to allow the option of sending prefetches across the memory bus normally or by a group prefetch communicator.

SPEC2000 Benchmarks

In order to show the behavior of our new implementation over a range of commonly used applications, we chose to run five different SPEC2000 benchmarks: gzip, mcf, parser, vpr, and apsi. The gzip benchmark is a compression algorithm that has five components: a TIFF image, a web server log, a program binary, random data, and a source tar file. Each input is compressed and decompressed, then compared to the original data to insure accuracy. Mcf is a benchmark used for combinatorial optimization. Based on a program used for single-depot vehicle scheduling in public mass transportation, this benchmark takes in inputs about the vehicles' timetables and outputs and optimal schedule. The parser benchmark tests word processing effects, parsing English sentences based on link grammar. Given an input sequence of English sentences, the parser outputs an analysis of each. Vpr is an integrated circuit computer-aided design benchmark. The input includes the netlist of the circuit, the FPGA architecture in which the circuit is to be implemented, and the assigned placement within the circuit. Vpr then outputs the final circuit routing, as well as statistics and validity check results. Finally, apsi is a floating point benchmark that deals with weather prediction. It reads in a 112x112x112 array of weather data over 70 different timesteps, then outputs the probably weather patterns. [4]

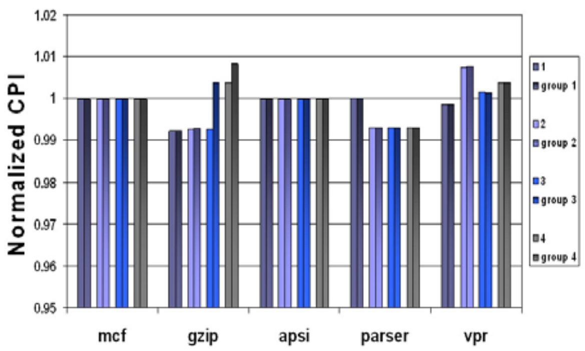
Experiments Run

To demonstrate the effect of our proposed architectural changes, we ran a number of experiments on each different benchmark. To first determine a base case for each, every test was run with all prefetching turned off. We then ran multiple simulations for each benchmark with the following variations: (1) Normal instruction prefetching turned on, using a prefetch width of 1, 2, 4, and 8. (2) Normal instruction and data prefetching turned on, using a prefetch width of 1, 2, 4, and 8. (3) Group instruction prefetching turned on, using a prefetch width of 1, 2, 4, and 8. (4) Group instruction and data prefetching turned on, using a prefetch width of 1, 2, 4, and 8. These experiments were designed to primarily show more efficient use of the memory bus resulting in a performance gain. Running the different benchmarks allowed us to see variations in our results due to the huge differences in existing applications. Once we had collected and studied the first set of data, we varied some architectural parameters and re-ran some of the simulations, a description of which can be found in later sections of this paper.

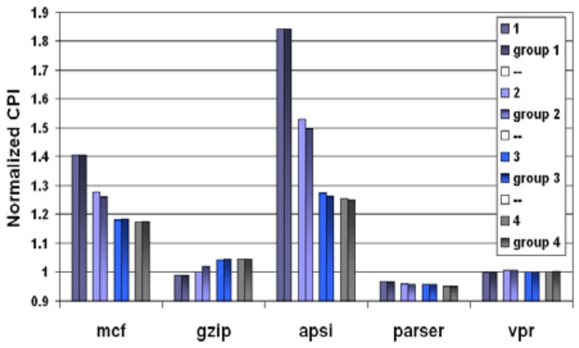
4 RESULTS

CPI

Figure 3 (a) compares the CPI of our five benchmark programs for normal next-n-line prefetching and group next-n-line prefetching for the Instruction Level 1 cache, while Figure 3 (b) compares the CPI for normal and group prefetching when prefetching is done for both the data and instruction Level 1 caches. In both figures the CPIs have been normalized to a base case in which no prefetching is activated. As the figures show, instruction prefetching has little impact on the overall CPI of the benchmarks and group prefetching does little to change this. Using both data and instruction prefetching actually detrimentally increases CPI, though group prefetching mitigates this increase to a small extent. There is a small exception for parser which shows slight gains with instruction and data prefetching, but this is more a result of normal prefetching rather than the use of group prefetching specifically.



(a) CPI Level 1 Instruction cache prefetching



(b) CPI with instruction and data prefetching

Figure 3: CPI with next-n-line prefetcher enabled. Each bench-mark was run with a prefetch width of 1, 2, 4, or 8. Each prefetch width was simulated both as individual and group requests

Coverage & Prefetch accuracy

The affect prefetching has on performance should be tied to how well it hides misses as hits on prefetches. As discussed before, this aspect of prefetch performance is described by the coverage factor of a prefetcher. Figure 4 shows the benchmarks with the best improvement in coverage factor for group prefetching over normal prefetching. Even these best case scenarios show little improvement in coverage with group prefetching, but this is to be expected given the previous illustration that CPI was largely unaffected. This small gain in coverage comes from prefetches arriving early enough to count as a hit. However if these converted misses originally only had small stall length, the overall change in CPI would be minimal. For example, if a miss under the baseline model is prefetched with normal prefetching from the L2 but the prefetch fails to return soon enough to change the miss into a hit, and with group prefetching the prefetch does return early enough to change the miss into a hit, then the maximum bus latency reduction by sending the prefetch as a group request is:

Reduction =
(On-chip bus latency) $\times$ (prefetch width -1)

With our baseline machine that has an on-chip bus latency of 1 processor cycle, this would be an extremely small gain.

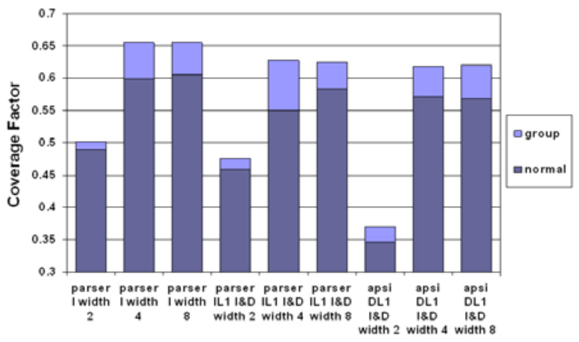


Figure 4: Greatest improvement in coverage factor using grouped prefetching over normal prefetching

The group prefetching method also left the amount of unnecessary and useless prefetches largely unchanged. The unnecessary prefetches are expected to remain mostly the same, as the prediction method (next-n-line) remained constant. Unnecessary prefetches would only go away when a missed line under normal prefetching was instead prefetched into a hit. This would not trigger a prefetch on the next N lines that had previously returned as unnecessary prefetches. The number of useless prefetches should only be increasing when the prefetcher has been altered to prefetch too quickly, resulting in prematurely evicted prefetches. This should result in a corresponding decay in the coverage factor. With the coverage factor remaining largely unchanged, a corresponding minimal change in the number of useless prefetches is expected.

Bus idle time

While the CPI and coverage properties of the prefetch were unaffected, it is still interesting to examine the effect

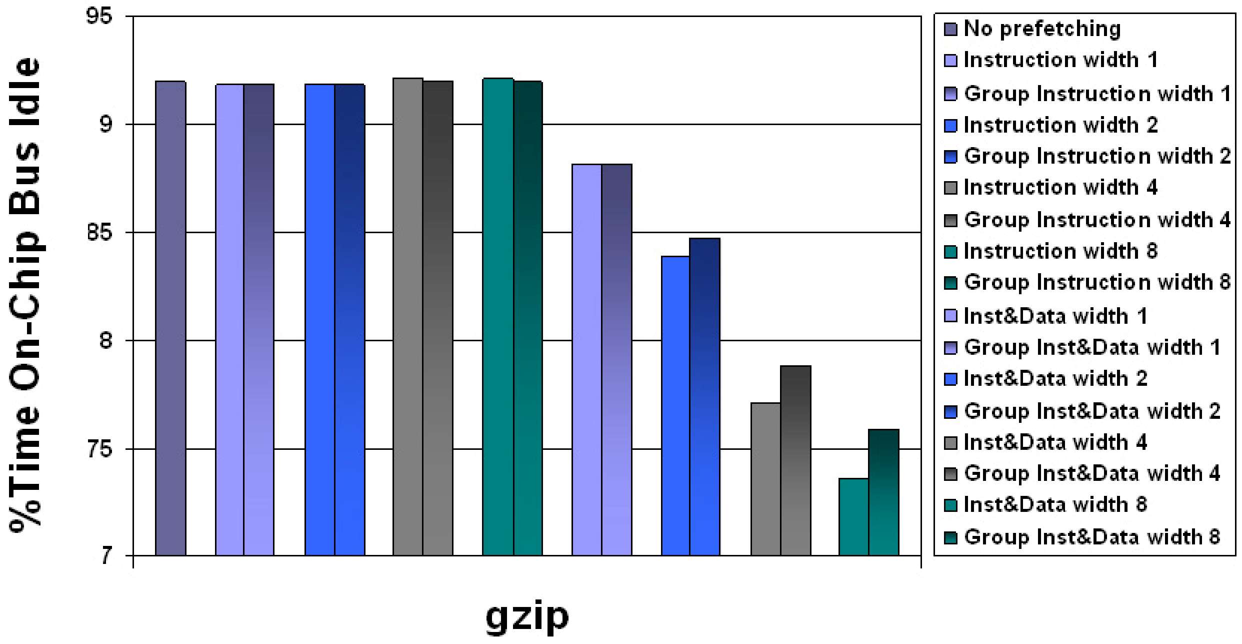


Figure 5: Percent of total cycles that the system bus was idle while running gzip

group prefetching has on bus usage. Figure 5 shows the percentage of cycles the on-chip bus was idle during the execution of gzip, the benchmark that showed the most improvement in on-chip bus usage. It should first be observed that in most cases the bus is idle for a significant portion of the cycles in the program. It is reasonable to assume that this indicates that memory stalls have a minimal affect on the benchmarks performance. Since the CPI is not changing and thus the number of cycle is remaining constant, the change in the amount of time the bus is idle can be inversely equated to the amount of time spent communicating on the bus. The high prefetch width instruction and data prefetching simulations indicate that group prefetching can improve the efficiency of bus usage. However, if the bus contention experienced is not significant enough to cause stalls, then increasing the efficiency of the bus usage will not have an effect on CPI.

The memory bus exhibited idle percentages of over 90 percent for three of the five benchmarks. For the mcf benchmark, the memory bus idle percentage increased for the larger width data prefetching cases, but this coincided with an increase in CPI. Group prefetching did not change the bus idle percentage when compared with normal prefetching.

For the apsi benchmark, however, the memory bus idle percentage for the baseline case (with no prefetching) and instruction-only prefetching case ranged from 0.18 to 0.22 percent. These extremely small percentages indicate that the memory bus is in near constant usage in this case. Figure 6 shows the effect of prefetching on the memory bus idle percentage of the aspi benchmark when instruction and data prefetching are both used. The addition of normal instruction and data prefetching causes an increase in CPI as shown in Figure 3 (b), but also results in a much more significant increase in processor idle time. The change from normal prefetching to group prefetching causes a slight decrease in CPI, and also causes a further increase in the memory bus idle percentage. The high bus usage in the baseline case and effect of group prefetching on bus usage would seem to indicate that the high data use of aspi would be able to benefit from group prefetching when the memory system performance is degraded with respect to the CPU performance.

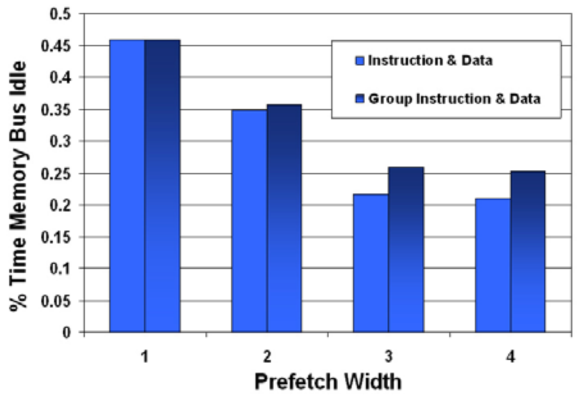


Figure 6: Percent of time that the memory bus was idle while running apsi with both normal and grouped instruction and data prefecthing

Artificial Constraints

Having examined the affect of group prefetching on an actual processor, one of the most interesting results is that group prefetching does have potential to improve bus usage. The question then becomes whether or not there are conditions under which this more efficient bus usage actually results in a performance improvement. We selected apsi as the benchmark to perform this test because its main memory bus was already close to 100 percent usage.

First we attempted to increase bus traffic by halving the size of the baseline L1 caches to cause evictions that require servicing. However, the spec2000 benchmarks memory footprints were too small for this to cause a change in the results. Second, the L1 caches were kept at the base line size, the L2 cache was reduced to 64 Kilobytes (from the original 2Mb) by reducing the number of sets to 1024, and the DRAM was altered by changing the CPU/DRAM clock ratio from 6 to 15. This still failed to result in a case where group prefetching generated a CPI improvement over a similar setting with no prefetching. Next, a third test was run using the same cache size and dram speed as the second test, but with the on-chip bus latency increased to 4 CPU cycles and the memory bus latency increased to 16 CPU cycles. This additional increase in bus latency finally resulted in a case where an increase in CPI was realized. A fourth test was run with the baseline memory sizes and DRAM speed and the only alteration being the increased bus latencies used in test three. However, this test failed to exhibit an increase in CPI for group prefetching over a similarly configured non-prefetching base case.

The third case with a decreased L2 cache size, decrease memory speed, and increased bus latency can be further examined to determine why group prefetching causes a decrease in CPU for this constrained system. Figure 7 shows the CPI of this test case normalized to a non-prefetching system with the same constrained memory setting. This figure shows that CPI decreased measurably only when both instruction and data prefetching were used, and that using group prefetching with both level 1 caches using prefetching can reduce CPI further for certain prefetch widths. The CPI didn’t change when only instruction prefetching was used. The coverage factor of the Level 1 Instruction cache does not change with the change to group prefetching, though it does increase with prefetch width as shown in Figure 8. The figure does however show that change from normal prefetching to group prefetching results in a large increase in the coverage factor of the data level 1 cache. Figure 9 show the change in queuing delay for the on-chip bus and memory bus normalized to the delay for a non prefetching processor using the same constrained memory system. The queuing delay remains constant for the cases with just instruction prefetching regardless of whether group prefetching is used. When data prefetching is added, the queuing delay for the on-chip bus increases, and the queuing delay for the memory bus decreases. Furthermore, adding group prefetching to data prefetching decreases the queuing delay on both buses, which results in the decrease in CPI using data prefetching. Finally, the number of replacements in the instruction level 1 cache range from 588 in the base case to 1452 with 8 wide instruction and data prefetching. In contrast the number of replacements in the data level 1 cache range from 3,125,858 to 3,151,021.

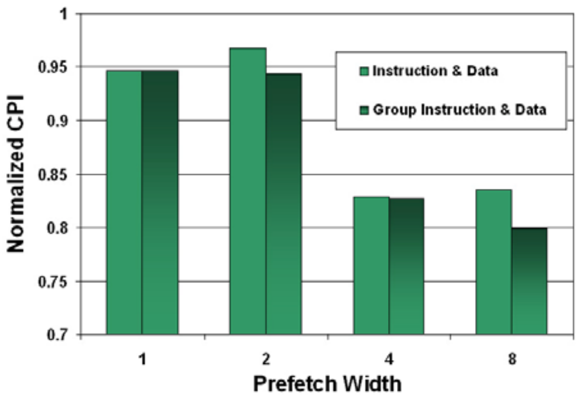


Figure 7: Normalized CPI for apsi with both normal and grouped instruction and data prefecthing

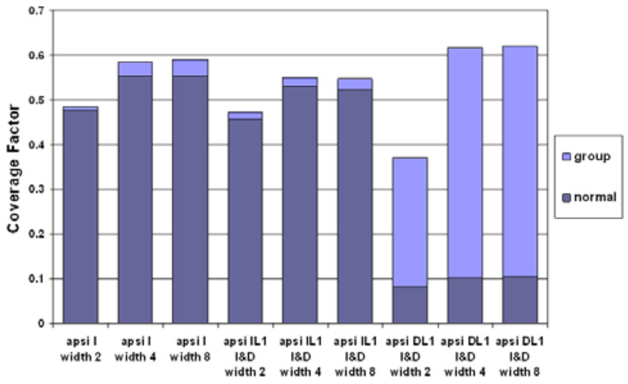


Figure 8: Coverage factor of L1 instruction cache remains unchanged with group prefetching.

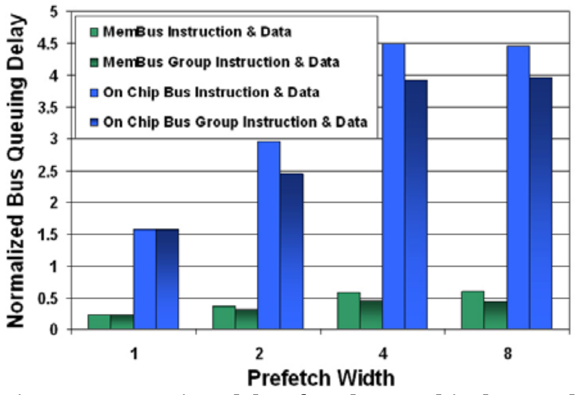


Figure 9: Queuing delay for the on-chip bus and memory bus normalized to the delay for a non-prefetching processor using a constrained memory system

For aspi, the servicing of data cache misses dominates the memory system for this constrained memory test. This results in instruction prefetching having no impact despite a reasonable coverage factor, while the coverage factor of data prefetching results in a significant improvement to CPI. The added coverage factor that group prefetching provides by reducing bus queue latency further extends this CPI improvement over the normal prefetching case.

Hypothesis Evaluation

It has been found that group prefetching does reduce bus contention and queuing delay, but that this only translates into an overall performance gain when applications exceed size of the cache system and the response of the DRAM and memory buses are significantly slower in relation to CPU speed. It remains to be seen that if with larger or more data intensive benchmarks this performance gain would be realized with a memory system that more accurately represents modern processors.

Logic design and testing of the group prefetch communicator will be necessary to verify that a group prefetch communicator can be implemented without increasing the hit latency for caches. If this is not possible, this increase in latency must be included in future simulations to evaluate the overall performance affect of group prefetching.

For the test case where a performance increase was released with group prefetching, the additional gain over normal prefetching did not exceed 4.0 percent. The additional development, logic design, and die costs of adding group prefetching would have to estimated and then weighed against this limited performance boost in a limited application space.

5 CONCLUSIONS

As future processors continue to increase in speed faster then memory systems, bus contention may become an increasing problem. While group prefetching does help reduce bus contention, this is only relevant to overall performance when sufficiently large demands are placed on the bus. Larger and more complex applications than the SPEC2000 benchmarks may show that current applications do actually generate such bus demand in modern applications. Before this idea can be implemented, it is still necessary to verify any additional latency group prefetching would add to cache hit times and to obtain a cost estimate for the additional logic.

ACKNOWLEDGEMENTS

We would like to thank Paul Willman, the TA for ELEC 525, for being the loving, friendly, and helpful labbie would made this project possible. We would also like to thank those two associate professors who didn’t want to let us borrow their memory simulator code.

REFERENCES

[1] Luk Chi-Keung, Mowry Todd C. “Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors.” Proceedings of Micro-31, Nov. 30 - Dec. 2, 1998, Dallas, Texas.

[2] Smith A. “Sequential program prefetching in memory hierarchies.” *IEEE Computer*, 11(2):7–21, 1978.

[3] http://www.simplescalar.com/

[4] www.spec.org/cpu2000/

APPENDIX

Latency of integer register read	-issue:int_reg_lat	1
Latency of fp register read	-issue:fp_reg_lat	1
Integer issue queue size	-issue:int_size	20
FP issue queue size	-issue:fp_size	15
Reorder buffer size (<number of entries>)	-rbuf:size	80
Load queue size (<number of entries>)	-lq:size	32
Store queue size (<number of entries>)	-sq:size	32
Additional victim buffer latency	-cache:vbuf_lat	1
Number of entries in the victim buffer	-cache:vbuf_ent	8
Queuing delay enabled in memory	-mem:queuing_delay	1
Queuing delay enabled in buses	-bus:queuing_delay	1
CPU freq / DRAM freq	-mem:clock_multiplier	6
0 - openpage, 1 - closepage autoprecharge	-page_policy	0
Time between start of ras command and cas command	-mem:ras_delay	1
Time between start of cas command and data start	-mem:cas_delay	1
Time between start of precharge command and ras command	-mem:pre_delay	1
1 - single data rate. 2 - double data rate	-mem:data_rate	1
Width of bus from CPU to DRAM	-mem:bus_width	16
Delay in chipset for request path	-mem:chipset_delay_req	2
Delay in chipset in data return path	-mem:chipset_delay_return	2
Line predictor	-bpred:line_pred	0
Line predictor width	-line_pred:width	4
Way predictor latency	-way:pred	1
Branch predictor type {nottaken|taken|perfect|bimod|2lev|comb|21264}	-bpred	21264
21264 predictor config (<l1size> <l2size> <lhist_size> <gsize> <ghist_size> <csize> <chist_size>)	-bpred:21264	1024 1024 8 4096 4 4096 4
Size of st wait table (0 for no table)	-fetch:stwait	1024
Line predictor speculative update	-line_pred:spec_update	1
Branch predictor speculative update	-bpred:spec_update	1
Disable slotting and clustering	-issue:no_slot_clus	0
Adder for computing branch targets	-slot:adder	1
Whether to use static slotting	-slot:slotting	1
Early inst. retire enabled	-map:early_retire	1
Load traps enabled	-wb:load_trap	1
Different size traps enabled	-wb:diffsize_trap	1
Trap if two loads map to same MSHR target	-cache:target_trap	1
Trap if two loads map to same cache line but have different addresses	-cache:addr_trap	1
Stall for 3 cycles of map < 8 free regs	-map:stall	1
Use load use speculation	-load:spec	1
Number of blocks to prefetch on a icache miss	-prefetch:dist	4
Cache configuration	-cache:define	DL1:512:64:0:2:F:3:vipt:0:1:0:Onbus
Cache configuration	-cache:define	IL1:512:64:0:2:l:1:vivt:0:1:0:Onbus
Cache configuration	-cache:define	L2:32768:64:0:1:l:7:pipt:0:1:0:Membus
Flush caches on system calls	-cache:flush	false

Sets maximum number of MSHRs per cache	-cache:mshrs	8
Sets maximum number of MSHRs per cache	-cache:prefetch_mshrs	4
Sets number of allowable targets per mshr	-cache:mshr_targets	8
Bus configuration	-bus:define	Onbus:16:1:0:0:1:0:L2
Bus configuration	-bus:define	Membus:16:4:0:0:1:0:SDRAM
Memory bank configuration	-mem:define	SDRAM
Define TLBs	-tlb:define	DTLB:1:32:0:128:l:1:vivt:0:1:0:Onbus
Define TLBs	-tlb:define	ITLB:1:32:0:128:l:1:vivt:0:1:0:Onbus
Data TLB config, i.e., {<config>|none}	-tlb:dtlb	DTLB
Instruction TLB config, i.e., {<config>|none}	-tlb:itlb	ITLB
	-cache:addr_trap	0
Enable/disable traps due to loads and stores with different sizes to the same address	-wb:diffsize_trap	0
Enable/disable trap if two loads map to same MSHR target	-cache:target_trap	0
Trap if MSHRs are full	-cache:mshrfull_trap	0
Instruction fetch queue size(in insts)	-fetch:ifqsize	4
Number of instructions to fetch per access	-fetch:width	4
Number of discontinuous fetches per cycle	-fetch:speed	1
Instruction slotting width(in insts)	-slot:width	4
Mapping width(in insts)	-map:width	4
Integer inst issue width(in insts)	-issue:intwidth	4
FP inst issue width(in insts)	-issue:fpwidth	2
Commit width(in insts)	-commit:width	11
Number of integer clusters	-res:iclus	2
Number of integer ALUs	-res:ialu	4
Number of integer multipliers/dividers	-res:imult	4
Number of fp clusters	-res:fpclus	1
Number of fp ALUs	-res:fpalu	1
Number of fp multipliers	-res:fpmult	1
Minimum cross cluster delay	-res:delay	1
Frequency of simulated machine	-mach:freq	463000000
Return address stack size (0 for no return stack)	-bpred:ras	32
Initial value of line pred bits	-line_pred:ini_value	0
Number of integer physical registers	-reg:int_p_regs	41
Number of fp physical registers	-reg:fp_p_regs	41