International Journal of Parallel Programming, Vol. 34, No. 2, April 2006 (© 2006)
DOI: 10.1007/s10766-006-0006-1

A PAB-Based Multi-Prefetcher Mechanism

Alexander Gendler,[1,2] Avi Mendelson,[2,3] and Yitzhak Birk[1]

Aggressive prefetching mechanisms improve performance of some important applications, but substantially increase bus traffic and "pressure" on cache tag arrays. They may even reduce performance of applications that are not memory bounded. We introduce a "feedback" mechanism, termed Prefetcher Assessment Buffer (PAB), which filters out requests that are unlikely to be useful. With this, applications that cannot benefit from aggressive prefetching will not suffer from their side-effects. The PAB is evaluated with different configurations, e.g., "all L1 accesses trigger prefetches" and "only misses to L1 trigger prefetches". When compared with the non-selective concurrent use of multiple prefetchers, the PAB's application to prefetching from main memory to the L2 cache can reduce the number of loads from main memory by up to 25% without losing performance. Application of more sophisticated techniques to prefetches between the L2- and L1-cache can increase IPC by 4% while reducing the traffic between the caches 8-fold.

KEY WORDS: Prefetching; cache tag pressure; memory wall.

1. INTRODUCTION

1.1. Background

The increasing gap between the speed of processor logic and the effective memory access time causes the "memory wall" effect; i.e., memory efficiency governs the performance of many applications such as databases and graphics (memory bounded applications). Hierarchical memory,

[1]Electrical Engineering Department, Technion, Haifa 32000, Israel.
E-mail: birk@ee.technion.ac.il
[2]Intel® Design Center, Haifa, Israel. E-mail: {alexander.gendler, avi.mendelson}@intel.com
[3]To whom correspondence should be addressed.

large cache lines and sophisticated prefetching mechanisms are used with modern processors to improve the effectiveness of the memory subsystem. However, as the gap increases, better techniques are required.

The idea of prefetching, first introduced in 1982,[1] is to mask memory latency by predicting future accesses and issuing advance requests to bring data to lower-level memory, e.g., cache. Software-initiated prefetches are generated by the compiler and appear to the system as instructions; hardware prefetchers track the behavior of the system and inject new events (usually a micro operation that receives special treatment rather than an instruction). In either case, prefetching is non-blocking; i.e., the processor continues normal operation concurrently with the prefetching activity.

The increasing processor-memory speed gap has affected prefetching in two ways: (1) the prefetcher must be able to look farther ahead, predicting addresses hundreds or even thousands of cycles ahead of time, and (2) the importance of correct prediction (or, viewed differently, the misprediction penalty) is greater.

Being speculative in nature, prefetching is not free: software prefetch instructions consume instruction bandwidth, and both hardware and software prefetches consume bus bandwidth and require tag checks. False prefetches in particular constitute overhead.

The timing of prefetching is also important,[2] as premature prefetching may bump useful data out prematurely; in fact, the prematurely prefetched data may itself be bumped out before being demanded. If a prefetch operation is issued too late, i.e., a demand request is issued while the prefetch event is being served, the demand request is blocked until the prefetch operation is completed. Some systems are smart enough to match the (pending) return data for the prefetch operation with the pending demand request. Most systems, however, do not do that (due to complications in the hardware, creation of race conditions and the fact that the prefetched data is being written to the L2 cache whereas the demand is usually brought to the CPU and to the L1 cache), and issue a demand request for the same block. Ideal data prefetching is timely, useful, and introduces little overhead. The aim of any good prefetcher implementation is to be as close as possible to the ideal behavior.

This paper focuses on hardware prefetching schemes. These schemes add prefetching capabilities to a system without the need for programmer or compiler intervention, and with no changes to existing executables. Hardware prefetching mechanisms use run-time information for their decisions, which can make them very efficient. Some merely enforce simple rules (e.g., ask for the block following (or preceding) the one referenced in a demand fetch unless it is already in the cache); others act as "correlators" that attempt to identify specific access patterns and their

parameters (e.g., stride) and request data based on those. As the importance of prefetching increases, hardware prefetchers are becoming more aggressive, and some systems even employ multiple prefetchers concurrently.[3,4] This consumes significant die area, power and other system resources, in addition to the aforementioned overhead. We will show how the judicious, dynamic selective use of multiple prefetchers can increase performance while reducing overhead.

1.2. Related Work

Hundreds of prefetching-related papers have been published over the past two decades, and new techniques are constantly being proposed as the "memory wall" problem becomes more severe. This section describes several commonly used prefetching algorithms, and discusses those techniques that our work is based on or compared with. See Refs. 5 and 6 for a more comprehensive survey. Special prefetching policies have been proposed for specific kinds of data[7–9] or specifically for instruction prefetching.[10,11] Some works present enhancements of basic algorithms.[12–16] Our focus in this paper, however, is on general-purpose prefetching policies composed from basic prefetching policies.

1.2.1. Basic Prefetching Algorithms

A processor usually issues requests to the memory subsystem at single-word granularity. Nonetheless, common practice entails the fetching of an entire cache line upon miss. This is the most basic form of prefetching, and is usually not even referred to as such. In this section, we review several prefetching algorithms.

1.2.1.1. Next-Line Prefetching. This simple prefetcher[1] prefetches the next cache line beyond the one being accessed or being requested by a demand cache miss. For simplicity, most current next-line prefetches are only triggered by demand cache misses. The basic algorithm prefetches the "demand cache line address + 1", but more sophisticated ones[17] also prefetch the "demand cache line address + complement of least significant bit (LSB)", thereby also supporting reverse sequential access. Using the complement of LSB effectively doubles the cache line size. Next-line prefetching is well matched to scanning arrays or executing sequential code.

1.2.1.2. Stride prefetching. This is a natural extension of the next-line prefetcher.[17–20] The algorithm examines the stream of demanded addresses in order to detect fixed strides. If a stride is found, a prefetch operation is triggered for the access address + stride size whenever a cache miss occurs.

Often, there are several concurrent "active" strides; e.g., a program that adds two arrays and saves the results in a third array will create three different stride patterns with a stride step of 1. In order to be able to track different strides concurrently, current methods use an array of "potential strides". In order to decide which entry in the stride table to compare the current cache miss with, some techniques limit the size of the stride step: others find the closest "active" stride pointer, and others[21] restrict the use of any single stride pointer to a single memory page.

1.2.1.3. Reference Prediction Table (RPT) Prefetching. RPT[22] extends the previous algorithms by associating an instruction address with the access pattern that it tries to track. It employs a table, indexed by a subset of the bits of the program counter (PC). When a Load operation is issued by an instruction at address (PC) m, the hardware looks it up in the table. If the address matches a table entry, stride detection is attempted. Otherwise, a new table entry is created.

Whenever a stride is detected for a specific instruction address, a prefetch instruction is generated by the hardware whenever a miss is caused by that entry and whenever the address of the "next triggered address" does not exist in the cache.

1.2.2. Design Tradeoffs for Prefetching Algorithms

Many papers have discussed tradeoffs in prefetching algorithm design. In this section we describe some of them:

1. *Prefetch destination*: Most of the existing prefetchers prefetch data from the main memory to the L2 cache. Traditionally, this was done because the L1 cache was very small and prefetched data could cause significant pollution in the cache (replace live cache lines with speculative data that may not be used). As the size of the L1 cache increases, some new techniques propose to also prefetch data from L2 to L1. [3,17]
2. *Prefetch depth*: Most current prefetch algorithms prefetch only a single cache line when a stride is detected. Some algorithms suggest prefetching several cache lines (using the same stride) in advance in order to mask larger latencies.
3. *Prefetch trigger*: A trigger to the prefetcher can be generated whenever the cache (L1 or L2) is accessed, or only upon a cache miss. The tradeoffs here are between extra pressure on the Tag array of the cache and the benefit of the prefetching mechanism.
4. *Software vs. hardware prefetchers*: The use of software prefetchers instead of or alongside hardware prefetchers.

5. *Number of prefetchers*: It has been observed that different prefetchers are capable of handling different access patterns. Thus, some modern architectures implement different prefetching techniques and use them all concurrently (e.g., Intel P4[3] and PowerPC G5[4]).

This paper examines the effectiveness of the use of a very aggressive multi-prefetcher mechanism that combines the use of all the following techniques as building blocks:

- Next line: Always prefetch the (demanded line +1)
- Stride: Similar to Next line prefetcher, but supports strides of different lengths and only one per page is allowed. We use 120 entries.
- RPT: Similar to Stride, but attach the stride calculation to a specific address (PC) in the page. We use 120 entries.

2. OBSERVATIONS

In order to understand the behavior of aggressive prefetching mechanisms, we use this section to study various aspects of each of the proposed prefetchers (Next line, Stride and RPT) and their combination. Miss rate, overall performance enhancement (IPC), the extra pressure on the cache Tags, and the bus traffic are all evaluated.

2.1. Miss Rate Patterns

We examined the impact of activating each of these prefetchers individually, as well as that of operating all of them concurrently. Figure 1 depicts the L2 miss rate throughout the execution of a GZIP program, taken out of the SPEC2000[23] performance suite.

Each point on the graph represents a 1000-instruction sliding window. Some of the findings are no intuitive: (1) although using "no Prefetcher" generates the most misses most of the time, there are time intervals during which the use of a prefetcher(s) generates more misses; (2) concurrent use of all prefetchers does not always yield the lowest miss rate, although in most cases it significantly reduces it; (3) no single method dominates all others, even throughout the execution of a single application program; and (4) there is a partial overlap between the techniques, i.e., more than one prefetcher may be asking for a given block.

2.2. Overall Performance (IPC)

Figure 2 depicts the IPC improvements (relative to no prefetching) obtained with the aforementioned prefetching techniques. Due to resource

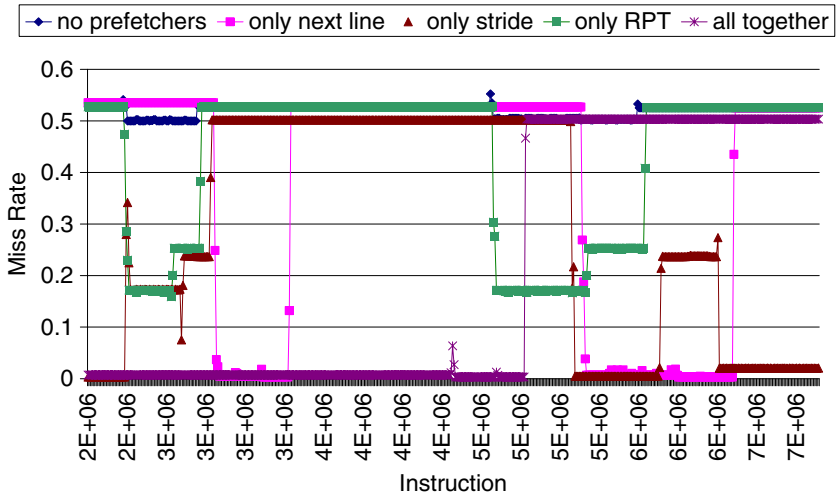

Fig. 1. Miss rate with various prefetcher combinations.

limitations, out of all the we choose to present results of six SPEC2000 benchmarks: GZIP, VPR, GCC, ART, MCF, and TWOLF. The benchmarks were selected to represent different variety behaviors in respect to data prefetching, starting from applications that significantly benefit from current prefetching mechanisms, such as ART, and ending with applications that prefetching has only small impact on them such as TWOLF. As can readily be seen, both the improvement itself and the relative improvement with different prefetcher combinations are highly application dependent. The exact simulation model that we used will be discussed in the next section.

The time- and application-dependence of relative performance of different prefetching techniques (Figs. 1 and 2, respectively) suggest that dynamic prefetcher configurations could yield substantial improvements.

2.3. Cache Access Traffic

One reason for the failure of some prefetching mechanisms to improve the overall performance is the amount of extra traffic they generate. In this section we examine the amount of extra traffic generated by each method; no-prefetching will serve as the baseline. We will also correlate these numbers with the IPC improvement that each of the prefetching algorithms yields.

Figures 3 and 4 depict the number of bus accesses without prefetchers and with all prefetchers activated. Figure 3 depicts the extra accesses to

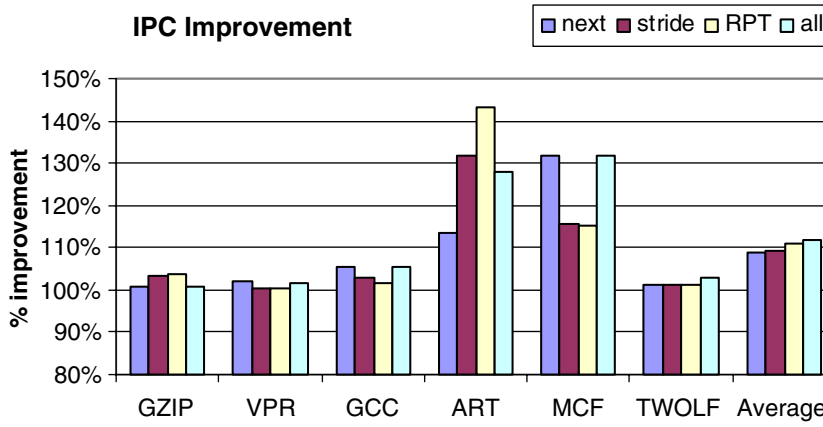

Fig. 2. IPC improvement for different Prefetchers.

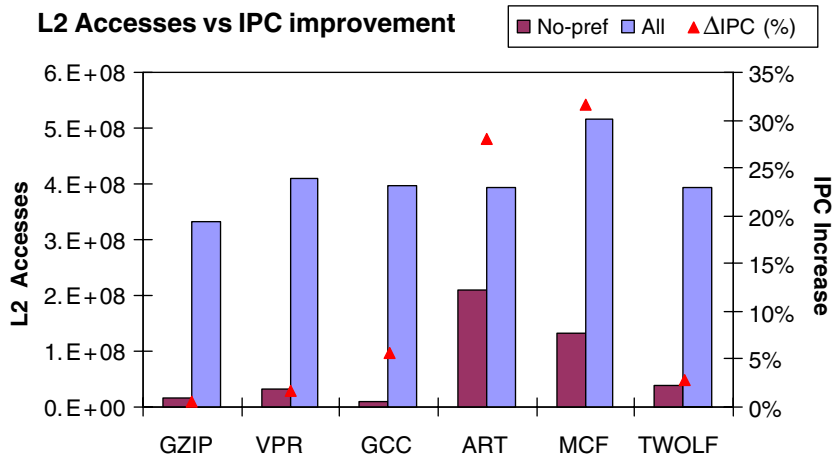

Fig. 3. Number of L2 accesses.

the L2 when we apply all the prefetchers to fetch information between the L1 and the L2, and Figure 4 depicts the extra traffic between main memory and the L2 cache.

From Fig. 3 it is evident that applying all prefetchers can cause very significant extra traffic between the L2 and the L1 caches. For applications like ART and MCF, the extra traffic can be justified by the vast improvement in IPC. However, for other applications, such as GZIP, the extra traffic contributes very little to the performance.

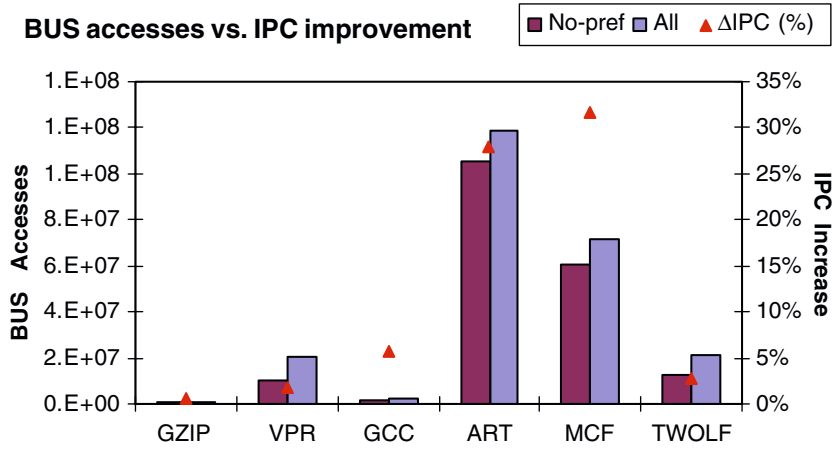

Fig. 4. Number of main-memory accesses.

Figure 4 depicts the main-memory bus activity with and without the prefetchers. It can be observed that some applications, such as GZIP and GCC (with the inputs that we chose for these simulations), do not generate many L2 misses, and apparently most of the prefetches found the information in L2, so few extra prefetch-related bus transactions are triggered. Other applications such as VPR, ART, MCF and TWOLF, which use larger working sets, generate significant extra bus traffic. Since bus load can significantly reduce overall system performance, it is very important to minimize unnecessary bus activity.

Due to the *concurrent* use of several prefetchers, different prefetchers sometimes generate requests to the same address. This may degrade performance. The solution is to coalesce such identical requests while they are waiting for service. At the L2 level, a request resides in the queue for a very short time, so the probability of the coalescing to occur is not high and so it does not significantly boost performance. Requests directed to main memory, in contrast, reside in the queue for much longer times. The probability of the coalescing to happen is therefore much higher, and so its benefit. Our simulations support this observation and show that coalescing is much more significant and causes a major reduction in extra accesses to the bus in this case. Note that the impact of saving requests to main memory depends on the utilization of the bus. If the bus is not loaded, the impact of the extra traffic on the bus may not be very noticeable. At times, however, the external bus is shared among several agents (not modeled in our simulation), in which case saving bus traffic may be very significant. Our scheme uses coalescing. Moreover, in order to isolate the contribution of our judicious use of multiple prefetchers, the configurations with which

we compare our scheme are also augmented to employ a coalescing mechanism.

2.4. Summary of the Main Observations

The observations presented in this section indicate that:

1. Although each prefetcher aims to capture a different access pattern, at execution time there are segments of the execution where a significant overlap between the requests issued by different prefetchers is observed. In other segments, different prefetchers behave differently.
2. No single prefetcher or prefetcher combination dominates all others throughout the entire execution of a single application program.
3. No single prefetcher or prefetcher combination (averaged over application execution) dominates all others across applications.
4. There is no direct correlation between the amount of prefetch accesses that were generation and the performance improvement they can achieve.

For some applications, the use of a single prefetcher may yield better performance than the use of a combination of all the prefetchers. Some applications show similar performance when running with prefetchers and without them.

3. PREFETCHER ASSESSMENT BUFFER (PAB)

The observations presented in the last section suggest that a dynamically adaptive hardware prefetching mechanism may offer significant advantages over current schemes. We now present such a mechanism, termed *Prefetcher Assessment Buffer*. The PAB enables tracking the success rate of the different prefetchers implemented as part of a given architecture and optimizes their use.

3.1. Possible PAB Structures

A simple implementation of the PAB entails keeping information regarding the N most recently prefetched cache line addresses. The information for each such cache line includes an indication whether the system actually accesses it. Based on this building block, different PAB structures can be constructed. For example:

1. Global PAB: The buffer contains information for all prefetchers' requests, without reference to the issuing prefetcher. It can be

good for analyzing global prefetcher performance. This non-specific information can only support a decision whether to operate all prefetchers or none of them.
2. Per-prefetcher PAB: Keep a different PAB for each prefetching algorithm being employed. Also, make it possible to compare performance of different prefetchers. This can support more sophisticated policies, e.g. "activate only the best-performing prefetcher".
3. Per-prefetcher.page PAB: Enables a performance comparison among prefetchers on a page-by-page basis. Based on this comparison one may, for example, elect to only use the best prefetcher for addresses in any given page at any given time. We will focus on this policy in the performance studies (Section 5).

Remark. PAB updates may continue even when the relevant prefetcher is not active.

In this paper we integrate the idea of page-based prefetch algorithms with the new PAB structure. A PAB entry is assigned on a per prefetcher.page basis. In the following experiments we can track at most 120 active pages at a time, with up to 16 PAB entries for each of them. Note that the prefetcher activation choice is made with a memory page granularity.

3.2. Policy

We examined various prefetcher-activation policies based on the information collected in the PAB. In this paper, however, the only dynamic policy that we evaluate entails the activation of only the best prefetcher at any given time for the page containing the triggering instruction. In order to implement the policy, the "Per-page PAB" structure, described in Section 3.1, is used. The *prefetcher hit ratio* of a given prefetcher is the fraction of requests issued by a that prefetcher that are subsequently matched by demand requests while still in the cache. Based on the Per-page PAB mechanism and the hit ratios of the various prefetchers, the best prefetcher is chosen at any given time for any given page. Only the chosen prefetcher is allowed to issue requests for any given page, but all other prefetchers also continue to operate (without sending out actual requests) in order to update their forecasts and hit ratios

4. SIMULATION SETUP & MODEL

The simulator used in this study was derived from the SimpleScalar 3.0 tool set[24]. We used the Sim-Outorder to drive an Out-of-Order

performance model of a 4-wide machine, but had to extend the model to enhance the accuracy of the memory subsystem. The basic configuration parameters used in the simulations are given in Table 1.

Simplescalar presents a straightforward implementation of the memory subsystem: for every memory access. It checks where the information is located and whether it causes a TLB miss; all the memory inquiries are done at once, so the system does not take into account more sophisticated events such as bus contention, queuing effects, etc. We extended the implementation of the memory subunits to include the above effects.

The original model of the SIM-OutOfOrder was extended to include four new queues, as depicted in Fig. 5. The queues are used to handle the requests to the appropriate level of memory. Consider, for example, instruction fetch. Instead of going directly to the L1 cache module and the TLB module, the request is placed in the ICAHCE-Queue and is sent for

Table I. SimpleScalar Parameters

Parameter	Value	Parameter	Value
Instr. queue	8	Instruction decode, issue and commit bandwidth	4 per cycle
Branch prediction	Gshare History 8 Table 1024	RUU size	128
BTB	4×512 entries	LSQ	32
L1 Instruction	Block size: 64 Bytes Sets: 64 Associativity: 4 Replacement: LRU. Access time: 1	L1 Data	Block size: 64 Byte Sets: 64 Associativity: 4 Replacement: LRU. Access time: 1
L2 Unified cache	Block size: 64 Bytes Sets: 1024 Associativity: 4 Replacement: LRU Access time: 8	Main memory access time	Latency: 200 Frequency: 5 Ports: 2
ITLB	Page size: 4 KB Sets: 16 Associativity: 4 Replacement: LRU Miss penalty: 1000	DTLB	Page size: 4 KB Sets: 32 Associativity: 4 Replacement: LRU Miss penalty: 1000
Integer ALUs	4	FPU	4
Integer dividers	1	F-DIV	1

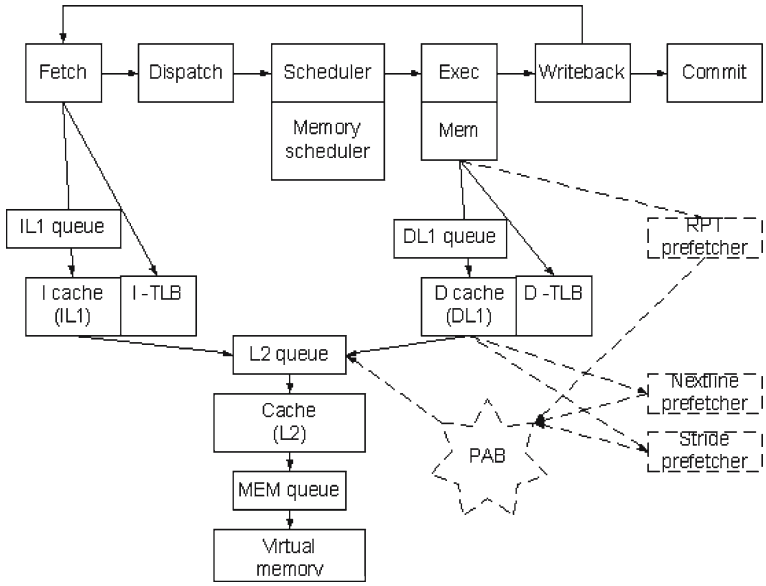

Fig. 5. Modified SimpleScalar Simulator.

service only once all the resources for the transaction are available. Similarly, in the case of an L1 cache miss, the request will be placed in the L2-Queue to be served based on system resources and schedule.

Load and store operations do not access the L1 Data Cache directly as in the original version of Simplescalar. Instead, the request is placed into the L1 Data Cache queue that contains all the requests that the L1 data cache needs to serve, and schedules the service of the requests according to the memory type, number of requests, priority, etc. Note that if the request is scheduled immediately, we are not charging it extra time for this operation. In our example, the L1 Data Cache has two access ports, so it can serve up to two requests concurrently. This structure allows us to simulate other important memory related effects as well. For example, we can distinguish the time it takes to bring the critical chunk of data to the CPU in order to allow its operation to continue from the time the buses are busy transferring the three remaining chunks to fill the buffer. The latter causes the port to be busy for a total of access time plus three core cycles from access time.

If the queue is full and cannot accept any new requests, it prevents the load/store unit from sending new requests until at least one entry is released. The same mechanism works for all levels of the memory

hierarchy. The same queues are also used to handle prefetch instructions. When necessary, the hardware sends a prefetch instruction to the proper queue. If the queue is full, the request is dropped; if the same line address already exists in the queue, the two requests are coalesced.

This new implementation allows us to better understand the impact of cache structure, widths of buses, and other configuration parameters on the overall performance (in terms of IPC) of the system. It also allows us to accurately model hardware prefetcher activities, by inserting them into the right queues and capturing the fact that prefetches are non-blocking, speculative operations that may be executed in any order or even dropped altogether without compromising correctness.

5. PERFORMANCE AND TRAFFIC OVERHEAD TRADEOFFS OF PAB-BASED PREFETCHERS

This section assesses the efficiency of the proposed PAB structure. In order to do that, we added to the simulator a PAB structure, and evaluated it with different mechanisms. For all the experiments in this section, we assumed the following PAB structure: a 16-entry cyclic buffer per page per prefetcher. Offering the highest resolution, this enables the best results in terms of IPC and L2/Memory traffic.

In the experiments whose results appear below, prefetching was triggered by all accesses to the L1 cache. Whenever a prefetcher identifies an opportunity for prefetching based on its own algorithm, it generates a request to the L2. If not found in L2, the request is forwarded to the bus in order to prefetch it from the main memory. Results for the case in which prefetching is only triggered by the L1 miss stream are presented in Ref. 25.

For each prefetcher, we added control logic that, based on the PAB statistics, decides whether the prefetch request will be sent to the L2 cache at all. If the PAB shows that a particular mechanism is not effective at a given time, its requests are ignored (but statistics-gathering continues). If an indication is given that we begin losing prefetching performance-enhancement opportunities, the logic resumes that prefetcher's activity.

The results presented in this section are mostly for the use of all three prefetchers with and without PAB control. Results for fewer prefetchers are presented in Ref. 25.

5.1. IPC Improvement with PAB-Based Mechanisms

Figure 6 depicts the IPC improvement for individual prefetchers, dynamic selective use of all prefetchers ("with control"), and non-selective

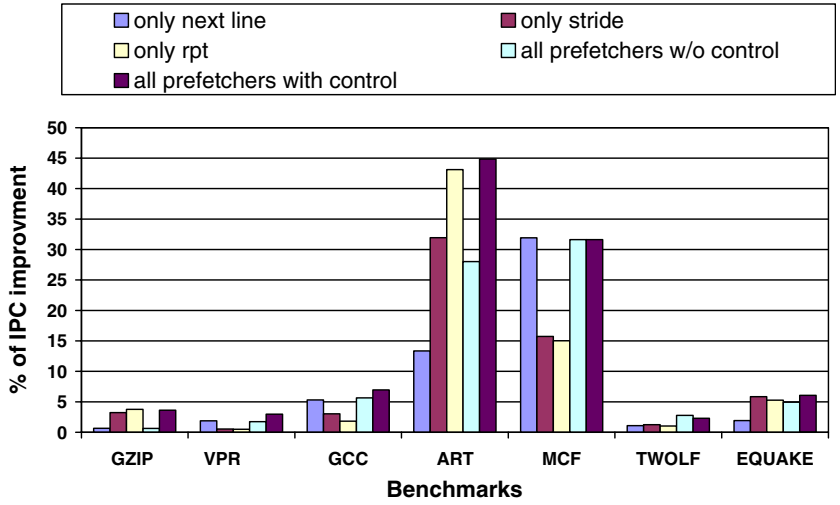

Fig. 6. IPC and L2 accesses for "L1 accesses" trigger.

concurrent use of all prefetchers ("w/o control"). Unlike previous results, wherein different prefetchers were best for different applications, we see that "all prefetchers with dynamic control" performs at least as well as any single prefetcher for all applications.

Figure 7 depicts the number of L2 accesses. For some benchmarks, the number of L2 accesses when the PAB is used to dynamically decide which prefetchers to use ("all prefetchers with control") is not much larger than with no prefetching; for others, the number of accesses is larger by up to 50%. In all but one case, the savings brought about by the use of the PAB relative to the "static" concurrent use of all prefetchers are dramatic.

It is furthermore interesting to note that applications such as GZIP, which hardly benefit from the aggressive prefetching mechanism, do not suffer from the vast L2 access stream, while applications such as ART, which significantly benefit from the aggressive prefetching mechanism, continue to incur a significant increase in L2 accesses. Thus, applications that benefit more also pay more.

The reduction in main-memory accesses is less pronounced, as seen in Fig. 8. Here, applications that do not benefit from the aggressive prefetching mechanisms do not generate vast traffic on the memory bus in the first place, so there is little they could gain. However, applications such as ART that do generate much traffic on the bus, show a significant gain in both IPC as indicated by Fig. 6 and L2 accesses (Fig. 7). Results for

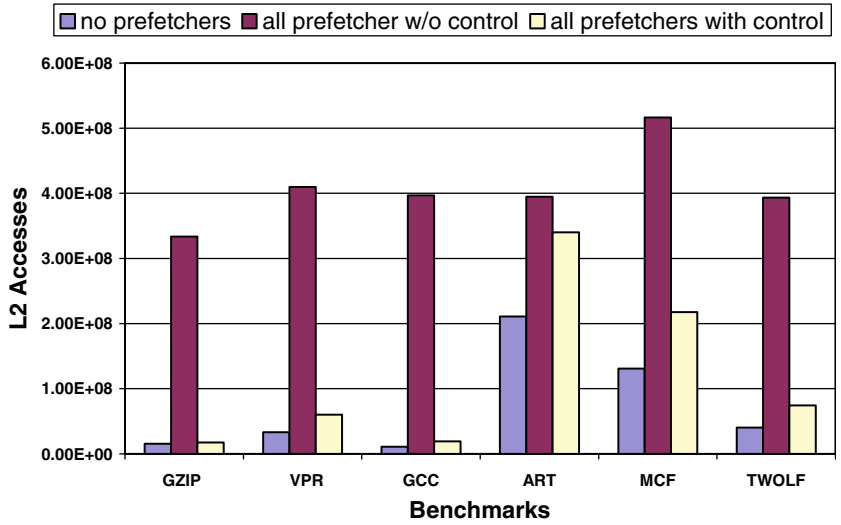

Fig. 7. L2 accesses.

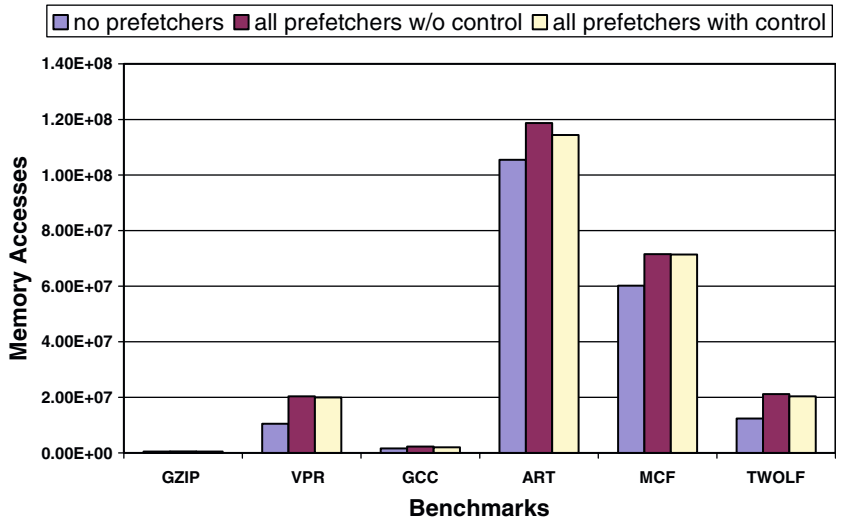

Fig. 8. Main memory accesses.

MCF suggest that more sophisticated control algorithms may improve performance and reduce traffic even further, since the IPC gain was achieved without reducing the extra overhead to the main memory (in comparison with no prefetching).

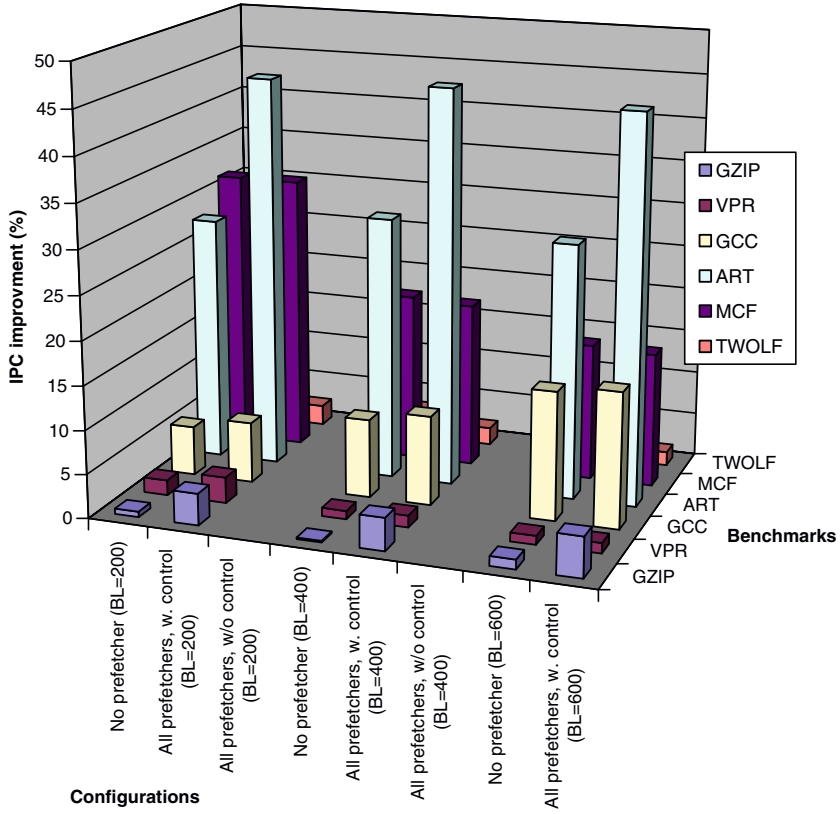

Fig. 9. IPC vs. BUS latencies (The numbers in parentheses are the latencies, expressed as the number of core cycles.)

Figure 9 presents the dependence of the IPC improvement on the main-memory bus latency. In addition to the value of 200 core cycles that is used throughout the paper, results are presented for 400 and 600 cycles. Results are provided for the cases of no prefetching as well as those with three prefetchers with and without control. The simulated bus is a split-transaction bus, which mitigates the effect of latency relative to that with a bus that does not allow transaction splitting. Indeed, the IPC improvement is nearly independent of bus latency.

6. SUMMARY AND CONCLUSIONS

This paper introduced the PAB along with related Control Logic as a new technique for efficiently and dynamically combining multiple

prefetchers. The performance of individual prefetchers, even when inactive, is constantly assessed (in terms of subsequent demand requests for prefetched lines), and this serves as the basis for deciding whether to enable any given prefetcher at any given time. We allowed the decision to be dynamic in time and to be made on a per-page basis, and also coalesced new requests with pending ones for the same cache line (both in our schemes and in the baselines used for comparison).

Simulation results show that restricting prefetching of data from any given page to a single prefetcher drastically reduces prefetch access rate and, with some additional techniques, retains the performance benefits in terms of IPC.

The non-selective concurrent use of multiple prefetchers can improve processor performance in terms of IPC. However, the multiple prefetch requests add a huge number of unnecessary requests that are cache hits by the time they are served and do not contribute to performance. The resulting extra load on cache lookup ports not only dissipates power; it also causes the loss of some of the performance improvement. The use of our PAB-based systems sharply mitigates these problems for some benchmarks, and eliminates them altogether for others.

Server systems usually disable prefetching in order to reduce the load on their cache lookup port, which is busier than those of desktop systems due to snooping activity. The use of PAB-based prefetchers in server systems would yield the benefits without the penalty.

Finally, unlike most current systems, PAB-based multi-prefetcher systems allow the beneficial use all L1 cache accesses as triggers for prefetchers, because PAB logic prevents most of the unnecessary prefetch requests.

This paper focused on the prefetching policies, with limited attention to implementation. The encouraging simulation results warrant further research into efficient hardware designs for the PAB and the related logic, and the consideration of PAB-based prefetching for inclusion in future products.

REFERENCES

1. A. J. Smith, Cache Memories, *Computing Surveys*, **14**(3):473–530 (Sep. 1982).
2. T. F. Chen and J. L. Baer, An Effective On-chip Preloading Scheme to Reduce Data Access Penalty, *Proceedings of Supercomputing 91*, pp. 176–186 (Nov. 1991).
3. Intel Co.: Intel Pentium 4 and Intel Xeon Processor Optimization, http://cache-www.intel.com/cd/00/00/01/76/17672_24896607.pdf or developer.intel.com.
4. Apple Computer, Inc., Technical note TN2087: PowerPC G5 Performance Primer, http://developer.apple.com/technotes/tn/tn2087.html.
5. S. P. Vanderwiel and D. J. Lijia, Data Prefetch Mechanisms, *ACM Computing Surveys*, **32**(2): 174–199 (Jun. 2000).

6. J. Tse and A. J. Smith, CPU Cache Prefetching: Timing Evaluation of Hardware Implementations, *IEEE Transactions on Computers*, **47**(5):509–526 (May 1998).
7. C. J. Hughes and S. V. Adve, Memory-Side Prefetching for Linked Data Structures, *Technical Report UIUCDCS-R-2001–2221*, Univ. or Illinois UC (May 2001).
8. M. Karlson, F. Magnus, and P. Stenstrom, A Prefetching Technique for Irregular Accesses to Linked Data Structures, *Proceedings of the 6th International Symposium on High-Performance Computer Architecture* (*HPCA*), pp. 206–217 (Jan. 2000).
9. N. Kohout, S. Choi, D. Kim, and D. Yeung, Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism For Pointer-Chasing Codes, *Proceedings of yhe International Conference on Parallel Architectures and Compilation Techniques*, pp. 268–279 (Sep. 2001).
10. J. C. Chiu, S. A. Chi, and C. P. Chung, Instruction Cache Prefetching Directed by Branch Prediction, *IEE Proceedings – Computers & Digital Techniques*, **146**(5):241–246 (Sep. 1999).
11. C-K. Luk and T.C. Mowry, Architectural and Compiler Support for Effective Instruction Prefetching: A Cooperative Approach, *ACM Transactions on Computer Systm*, **19**(1):71–109 (Feb. 2001).
12. G. S. Manku, M. R. Prasad, and D. A Patterson, A New Voting Based Hardware Data Prefetch Scheme, *Proceedings of the 4th International Conference on High Performance Computing*, pp. 100–105 (Dec. 1997).
13. T. L. Johnson, D. A. Connors, and W-M.W. Hwu, Run-time Adaptive Cache management, *Proceedings of the 31st Hawaii International Conference on System Sciences*, vol. 7, pp. 774–775 (1998).
14. A. Ki and A. E. Knowles, Adaptive Data Prefetching Using Cache Information, *Proceedings 11th International Conference on Supercomputing*, pp. 204–212 (1997).
15. R. Pendse and H. Katta, Selective Prefetching: Prefetching when Only Required, *Proceedings of the 42nd Midwest Symposium on Circuits and Systems*, vol. 2, pp. 866–869 (Aug. 1999).
16. P. Reungsang, S. K. Park, G. Lee, S. -W. Jeong, and H. -L. Roh, Reducing Cache Pollution of Prefetching in a Small Data Cache, *Procedings of the International Conference on Computer Design* (ICCD), pp. 530–533 (Sep. 2001).
17. IBM, POWER-3: Next generation 64-bit PowerPC Processor Design, www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power3wp.html.
18. A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez, Toward Kilo-Instruction Processors, *ACM Transactions Architech and Code Opt.*, **1**: 289–417 (2004).
19. J. W. Fu, J. H. Patel, and B. L. Jansen, Stride Directed Prefetching in Scalar Processors, *Proceedings of the International Symposium on Microarchitecture* (*MICRO-25*), pp. 102–110 (Dec. 1992).
20. I. Sklenar, Prefetch Unit for Vector Operation on Scalar Computers, *Proceedings 19th International Symposium on Computer Architecture* (ISCA), pp. 31–37 (May 1992).
21. H. Yu and G. Kedem, DRAM-page Based Data Prediction and Prefetching, *Proceedings International Conference on Computer Design* (ICCD), pp. 267–275 (Sep. 2000).
22. T. F. Chen and J. L. Baer, Effective Hardware-Based Data Prefetching for High Performance Processors, *IEEE Transaction on Computers.* **44**(5): 609–623 (May 1995).
23. J. F. Cantin and M. D. Hill, Cache Performance for SPEC CPU2000 Benchmarks, http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/.
24. T. Austin, E. Larson, and D. Ernst, SimpleScalar: An Infrastructure for Computer System Modeling, *IEEE Computer*, **35**(2): 59–67 (Feb. 2002).
25. A. Gendler, Efficient Multi-Prefetcher Architectures using a Prefetcher Assessment Buffer (PAB), MSc Dissertation, Electrical Eng. Dept., Technion (2005).