

# Pointer-Based Prefetching within the Impulse Adaptable Memory Controller: Initial Results

Lixin Zhang, Sally A. McKee, Wilson C. Hsieh, and John B. Carter

*Department of Computer Science*

*University of Utah*

{lizhang, sam, wilson, retrac}@cs.utah.edu

<http://www.cs.utah.edu/impulse/>

## Abstract

*Prefetching has long been used to mask the latency of memory loads. This paper presents results for an initial implementation of pointer-based prefetching within the Impulse adaptable memory controller. We conduct our experiments on a four-way issue superscalar machine. For the microbenchmarks we examine, we consistently realize about a 20% improvement in execution time for linked data structures accessed within medium to short loop iterations. This compares favorably to software prefetching when the data working set fits in cache, and exceeds the performance of the latter technique for large working sets. We also find that a superscalar, out-of-order processor hides the memory latency of linked data structures accessed in large loop iterations exceptionally well, which makes any pointer prefetching unnecessary.*

## 1 Introduction

Prefetching has long been used to mask the latency of memory loads. This paper presents results for an initial implementation of pointer-based prefetching within the Impulse Adaptable Memory Controller system: whenever the memory controller sees a request for a node in a linked data structure, it prefetches all objects directly pointed to by the this node. Our results show that under some circumstances, traditional software prefetching is preferable to memory-controller based prefetching, but

in many situations, the memory-controller based scheme or a combination of the two approaches wins.

Prefetching can be performed by software, hardware, or a combination of both. The purely software approach relies on a compiler to generate instructions to preload data [23, 20], or an application writer to modify source code to achieve the desired behavior [3, 17, 25]. Hybrid approaches provide hardware support for such prefetch operations. For instance, they might augment the ISA with a prefetch instruction [10], redefine a load to a specific register (e.g., to register 0, as in the PA-RISC architectures [15]), or provide programmable prefetch engines [6] or programmable stream buffers [19]. Hardware-only prefetching [2, 9, 12, 14, 29] thus has the advantage of being transparent, and some commercial machines include such mechanisms [5, 7, 28]. However, due to its speculative nature, care must be taken to keep from lowering application performance by increasing contention in the caches and wasting bus bandwidth on useless prefetches.

Most prefetching research in the literature focuses on fetching data structures with regular access patterns, such as streams or arrays. Some of these require that stream patterns be detected dynamically, as in the vector prefetch units proposed by Baer and Chen [2], Fu and Patel [13], and Sklenar [29]. Cache-based approaches, such as the sequential hardware prefetching of Dahlgren *et al.* [9], eliminate the need for detecting strides dynamically. To minimize the number of unnecessary prefetches, the *prefetch distance* of these run-time techniques is generally limited to a few loop iterations (or a few cache lines). When prefetching to cache, one risks the possibilities that the prefetched data may replace other needed data or may be evicted before it is used.

Contention in the on-chip cache hierarchy can be avoided by buffering prefetched data lower in the memory system. For instance, the Stream Memory Controller

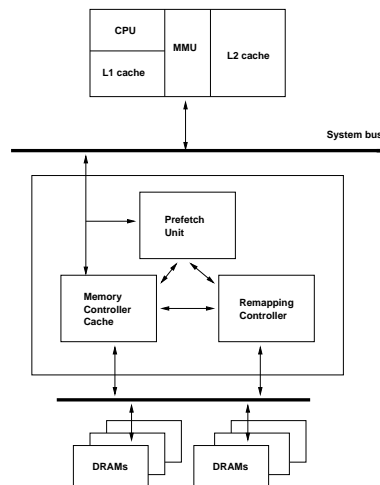
---

This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

of McKee *et al.* [19] combines prefetching within the memory controller with dynamic access ordering to exploit bank parallelism and locality of reference among the DRAM pages.

Other researchers propose to move the prefetching mechanisms all the way down to the DRAM chips. Alexander and Kedem demonstrate significant speedups for some benchmarks via on-chip DRAM prefetching and caching [1]. Their scheme modifies the DRAM structure to include an SRAM cache and a prediction table. The table stores addresses of up to four possible blocks that are likely to be accessed soon after a given block. When an access hits in the table, all the associated predicted blocks are prefetched into SRAM buffers inside the DRAM chip. Using a large prefetch block size keeps the prediction table small. Such DRAM-based prefetching and caching techniques exploit the large, available on-chip bandwidth without increasing bandwidth requirements of the bus between the memory controller and the DRAM chips. Wong and Baer [31] investigate line-size and associativity tradeoffs for cached DRAMs. They add a prediction mechanism to the memory controller to decide whether to close the page in the DRAM sense-amps after an access. Unlike memory-controller based prefetching and caching, DRAM-based approaches must still access the DRAM chips on a prefetch hit, and thus such schemes incur a few cycles higher hit latency than a memory-controller based scheme. Note that for any dynamic scheme that depends on history, memory latencies during the history-collection phase cannot be hidden.

Several recent CPU-based techniques address the prefetching of *linked data structures* consisting of nodes connected by pointers. Some of these [18, 21, 26] try to overlap the latency of a prefetch with the CPU activity between two consecutive accesses. When there is not enough work per node to mask the prefetch latency, these methods become less effective. Two other techniques [18, 27] add jump pointers, which point to nodes located further down a linked list. These pointers increase the prefetch distance by allowing non-adjacent nodes to be fetched for future iterations, which can hide load latencies even when there is little work per iteration. Karlsson *et al.* [16] extend this work to *prefetch arrays*; they add a new *block prefetch* instruction to the ISA. Prefetch arrays consist of a number of jump pointers located consecutively in memory. Both jump pointers and prefetch arrays consume extra memory and require non-trivial pointer pre-processing. To be effective, these techniques require that the traversal path through a data structure be known beforehand and that the data structure be traversed many times. The memory-controller-based approach we propose here requires no extra mem-



**Figure 1.** Organization of the Impulse Memory Controller.

ory or extra pointers, and it is capable of hiding latency even when there is very little work to do for each node.

## 2 Impulse Background

Impulse expands the traditional virtual memory hierarchy by adding address translation hardware to the memory controller (MC) [4, 11, 30, 33]. The operating system can configure the MC to reinterpret unused physical addresses as remapped aliases of real physical addresses. The remapped physical addresses refer to a *shadow address space*. This virtualization of unused physical addresses can be used to improve the efficiency of the processor caches. The operating system manages all the resources in the expanded memory hierarchy and provides an interface for the application to specify optimizations for particular data structures. The programmer (or the compiler) inserts directives into the application code to configure the memory controller.

The results presented in Section 5 focus on another aspect of the Impulse controller. Since DRAM access times constitute the majority of memory access latencies [8], Impulse prefetches data within the memory controller to mask these DRAM latencies. Figure 1 shows the block diagram of the relevant portions of the Impulse memory system. Prefetching is performed by the following components:

- a *Remapping Controller*, which includes control registers to store remapping information, a simple ALU (AddrCalc) to translate remapped addresses, and a *Memory Controller TLB* (MTLB) to cache recently used address translations.

- a *Prefetch Engine* that generates prefetch addresses;
- a small SRAM buffer (the *Memory Controller Cache*, or MCache) that buffers data prefetched from DRAM;

If a load request hits in the MCache, the memory controller can return the requested data sooner than if it had to fetch the data from DRAM. The MCache is physically indexed and tagged. Its line size equals the size of a system block, the data unit transferred between processor and main memory. The MCache uses a write-invalidate protocol. Data in the MCache can never be dirty, which means that a victim line can simply be discarded when a conflict occurs.

### 3 Prefetching in Impulse

Processor-based prefetching techniques do not necessarily adapt readily to MC-based or DRAM-based prefetching: some of the information available to the processor is lost to the lower portions of the memory system, since they only see physical addresses. For instance, some of the regularity of the address stream generated by the processor will have been lost in the virtual-to-physical translation. This absence of information makes it more difficult to design efficient MC-based prefetching mechanisms. Nonetheless, MC-based prefetching has several potential advantages over schemes that preload data onto the processor chip. MC-based prefetching does not add complexity to the processor, does not pollute the on-chip cache hierarchy, and does not decrease bus utilization. Furthermore, MC-based prefetching is portable, in the sense that changing from a uniprocessor to multiprocessor or even changing the CPU cache hierarchy requires no modifications to the prefetching mechanisms. This prefetching technique is independent of changes to the processor chip, since it resides within the memory controller on the other side of the system bus.

The price for this flexibility is that MC-based prefetching techniques cannot hide bus arbitration and transmission times, although modern latency-tolerant CPU designs may be able to help mask these bus-related delays. Fortunately, the non-DRAM component of main memory access times (i.e., the number of cycles required to arbitrate for the bus, transfer blocks of data, perform ECC, etc.) is roughly tracking processor speeds, while DRAM cycle times are falling behind. The relative performance payoff from MC-based prefetching should therefore grow in the future. Another possible source of latency with this technique comes from competition for memory banks and for the memory controller’s internal bus.

The pointer-prefetching technique we examine is quite simple: whenever the memory controller sees a request for a node in a linked data structure, it prefetches all objects directly pointed to by this node. The brute-force approach taken in these initial studies thus leaves much room for refinement, but requires minimal hardware support to determine which fields are to be prefetched.

To support pointer-chasing, the MC must be able to:

- identify accesses to nodes in linked data structures,
- identify and dereference pointers, and
- translate virtual addresses to physical addresses.

The first step identifies accesses to linked objects. We remap the contiguous virtual region that contains all the linked objects to a contiguous region in the “shadow address space” composed of the (previously) unused physical addresses. Any address lying inside this shadow region is interpreted by the MC to be an access to a link node. When the MC recognizes such an address, it prefetches the objects pointed to by the referenced node. Providing the memory controller with information about linked nodes is facilitated by a special *malloc()* used exclusively for linked objects. If a linked object resides on the stack or in statically allocated space in the original program, the programmer or compiler must change the program to use dynamically allocated memory from the heap. The OS reserves a large virtual region for the special *malloc()* and maps it to a shadow region. Note that this process wastes no real physical memory, since the shadow addresses are not backed by DRAM. The special *malloc()* then uses virtual addresses inside this region and allocates real physical memory as requested. This scheme places all the linked objects (and only those objects) inside the virtual region reserved for this purpose.

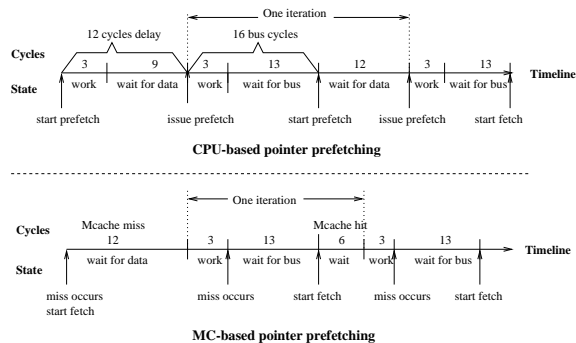
The second step obtains the addresses of linked objects pointed to by the requested object. To compute the addresses of pointer fields, the MC must first recover the starting address of the requested object. For a shadow address *saddr* inside a region [*saddr\_start*, *saddr\_end*), the starting address of the requested object is (*saddr* - *saddr* % *object\_size*). If linked nodes are a power of two in size ( $2^x$ ), the starting address is simply *saddr* with the least *x* significant bits cleared. The MC now needs to know which fields contain pointers. We investigate two methods for identifying pointer fields — using a bit mask to indicate which words are pointers, or representing the pointer-field offsets directly. Each method has limitations on the object size and the number of pointer fields, depending on how many bits are devoted to the representations. In general, the first method can identify more fields, whereas the second permits bigger objects.

We currently assume that no object is larger than 128 bytes. When a linked node exceeds this limit, we represent it as a collection of appropriately sized smaller objects. In our experiments in Section 5, we use a 32-bit register to maintain pointer location information. If there are fewer than five pointer fields, we use direct offsets, with one byte describing each pointer. Otherwise, we use a bit mask.

The third step translates the pointers fields from virtual to physical addresses. If the pointer addresses fall inside the remapped virtual region, the MC translates them first to shadow addresses and then to physical addresses via the remapping controller, and finally it issues the physical addresses to DRAM. When the prefetched data arrives, it is stored into the MCache.

Figure 2 illustrates the differences between CPU-based and MC-based pointer prefetching. To see why these differences arise, consider what happens for a typical prefetch operation under each approach. In the case of CPU-based prefetching, the program usually issues a prefetch to the next node while it works on the current node. It then must potentially wait for the next node to return before it can continue. After the data returns, it issues another prefetch and continues processing. Whenever the processor requests a block, the memory controller fetches it from DRAM and sends it back on the system bus. In a system with two to four times more bandwidth inside the memory controller than the system bus, fetching a cache line from DRAM takes two to eight cycles; sending a line over the system bus takes eight to sixteen cycles. This means that in MC-based prefetching schemes, the controller can begin accessing the next nodes immediately after the requested pointer has been fetched, even before it completes the bus transfer. Since systems usually contain several independent DRAM banks, the MC can probably prefetch several lines into the MCache by the time the system bus completes sending the requested line. Even if the processor requests a prefetched node shortly after the previous one returns, the MC should be able to provide the data quickly.

Figure 2 assumes that the memory system returns the critical word first, uses 128-byte blocks, and has an 8-byte wide bus. After a cache miss occurs, the critical word returns in six bus cycles if it hits in the MCache, or in 12 bus cycles if it misses. This diagram assumes that the program spends three bus cycles working on each node: this is an aggressive assumption, in that it is difficult to prefetch that quickly. For CPU-based prefetching, we also assume that the first node has been prefetched successfully. Figure 2 shows that with MC-based prefetching the loop takes 22 cycles per iteration, whereas with CPU-based prefetching it needs 28 cycles.



**Figure 2.** MC-based pointer prefetching vs. CPU-based pointer prefetching. *issue prefetch* means a prefetch instruction is executed. *start prefetch* means a prefetch is being sent to the bus.

## 4 Simulation Environment

Our studies use the execution-driven simulator URSIM [32] derived from RSIM [24]. URSIM models a MIPS R10000 microprocessor [22], a split-transaction MIPS R10000 cluster bus with a snoopy coherence protocol, and the Impulse adaptable memory system. The processor is a four-way, out-of-order superscalar with a 64-entry instruction window. The TLB is single-cycle, fully associative, software-managed, and has 128 entries. The 32-kilobyte L1 data cache is non-blocking, write-back, virtually indexed, physically tagged, direct-mapped, and has 32-byte lines. The 256-kilobyte L2 data cache is non-blocking, write-back, physically indexed, physically tagged, two-way associative, and has 128-byte lines. The system supports critical word first, i.e., a stalled request can continue after the first critical quad-word has returned. The split-transaction bus multiplexes addresses and data, is eight bytes wide, has a three-cycle arbitration delay and a one-cycle turnaround time. The system bus, Impulse memory controller, and DRAMs have the same clock rate, which is one third of the CPU clock's. The load latency of the first quad-word is 16 memory cycles. An L1 cache hit takes one cycle, and an L2 cache hit takes eight cycles.

The MTLB is configured to be two-way associative, with 64 entries and a one-memory-cycle lookup latency. The memory controller cache is two-way associative and has 32 128-byte blocks. An MCache hit reduces memory latency by six memory cycles. The remapping controller takes one memory cycle to translate remapped addresses. The prefetch engine can dereference one pointer per cycle if the data is stored inside its buffer. The DRAM system has four banks, and each two banks share a 16-byte wide bus between the MC and DRAM.

To test the performance of MC-based pointer-chasing prefetching and to compare it with the equivalent CPU-based approach, we run a synthetic microbenchmark traversing nodes in a large linked list or tree:

```
ptrhead = (NODE *) malloc(...);
Initialization;
ptr = ptrhead;
while (ptr) {
    work(ptr);
    ptr = ptr->next;
}
```

The size of each node is 16 bytes for a singly linked list or 32 bytes for tree structures. All the linked nodes are first allocated as an array. To initialize the structure, each node is linked to the node(s) at the same position in the next system block (if a linked list) or blocks (if a tree). Traversing the list successively touches one node of each block, then repeats the procedure for other nodes. Each iteration of the microbenchmark’s major loop works on the node for a period of time, then moves to the next node in the structure. In our experiments, we vary the number of cycles spent working on each node to determine how iteration size will affect the performance of different prefetch algorithms. We generate software prefetching results by inserting prefetch instructions by hand at the top of the loop.

## 5 Results

The performance results we present for singly linked lists, binary trees, and ternary trees are obtained through complete simulation of the synthetic microbenchmark on URSIM. All execution times are normalized to the execution time of the microbenchmark with no prefetching. These times do not include initialization of the data structures. We give results for software prefetching and memory-controller based prefetching both in isolation and in combination.

Figure 3 shows the execution time for three different prefetching algorithms: CPU-based software pointer prefetching, MC-based hardware pointer prefetching, and a combination of both. CPU-based prefetching works well when prefetch requests hit in the L2 cache. The left half of Figure 3 shows that the CPU-based approach yields from 6% to 13% improvement for small iterations when the list contains 16K nodes. After accessing the first node of each block, all nodes are loaded into the L2 cache, but the L1 cache is too small to hold them all. Without software prefetching, subsequent accesses to other nodes in each block will miss in L1 and hit in L2. With software prefetching or the combined prefetching scheme, data is moved into L1 in advance, and so all these accesses hit. The combined approach delivers the best performance for all loop sizes.

When a prefetch has to go to memory, MC-based pointer prefetching yields better performance. For instance, if the program accesses only one node of each block, the CPU-based approach has no effect, but the MC-based approach improves performance by 17%, as shown by Figure 4(a). Surprisingly, the CPU-based approach has almost no effect when prefetch requests load data from the memory, no matter what the iteration size (shown in the right-hand graph of Figure 3). We traced the execution instruction stream to reveal that the superscalar, out-of-order processor hides memory latency exceptionally well. The assembly code of the major loop looks like:

	# without prefetch	# with prefetch
1:	ld ptr->value	ld ptr->value
loop:		loop:
2:		prefetch ptr->next
3:	<work on ptr>	<work on ptr>
4:	ld ptr->next, ptr	ld ptr->next, ptr
5:	cmp ptr,0	cmp ptr,0
6:	bne loop	bne loop
7:	ld ptr->value,...	ld ptr->value,...

The modeled processor in URSIM has a 64-entry instruction window and is able to issue four instructions each cycle. We assume <work on ptr> depends on ptr->value and instruction 1 results in a cache miss. While waiting for the data of instruction 1, the instructions following will be issued until the instruction window is full, even though these instructions cannot execute until the ptr->value data is available.

For small iterations, instruction 7 can be issued before instruction 1 loads its data. When instruction 1 gets data, <work on ptr> will continue execution immediately. In theory, instructions 4 to 7 can also continue immediately, because they do not depend on anything in <work on ptr>. The only thing that can stop them is unavailability of functional units (the address generation unit in the MMU). Our results show that instruction 7 can issue a read request (to load the next node in the list) in three to five cycles after instruction 1 graduates. Issuing a load for the next node five cycles after an iteration starts essentially prefetches a node one iteration earlier. Putting a prefetch instruction (2) at the top of each iteration actually starts fetching the next node only three to five cycles early, not an entire iteration. So, for each small iteration, execution time is very close to the memory latency of a cache miss. The superscalar processor almost completely overlaps the cycles spent working on each node with memory activity. On the other hand, MC-based prefetching can successfully reduce the average memory latency by around 20%. As a result, it can effectively reduce the execution time of each iteration by the same percentage.

For large iterations, instruction 1 will have graduated when instruction 7 is issued. Since instruction 7 does not depend on any values generated during <work on

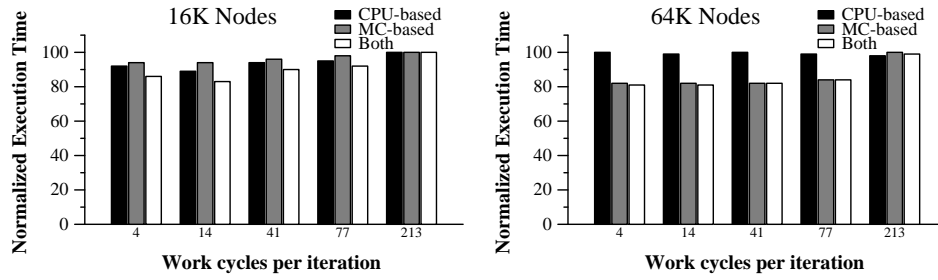


Figure 3. Normalized execution time for singly linked lists.

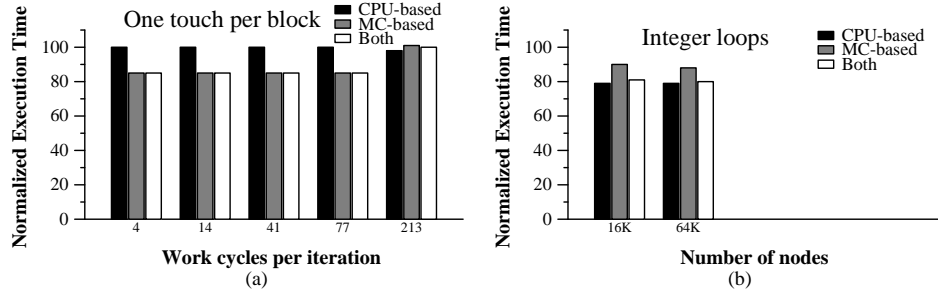


Figure 4. Normalized execution time for linked list traversal when (a) each block is touched only once, and (b) each iteration contains many integer and logical operations.

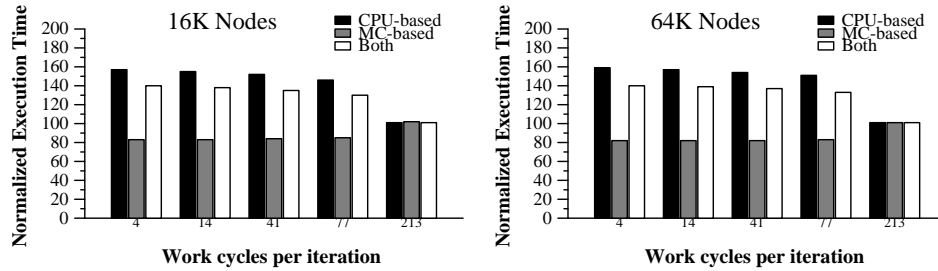


Figure 5. Normalized execution time for binary tree.

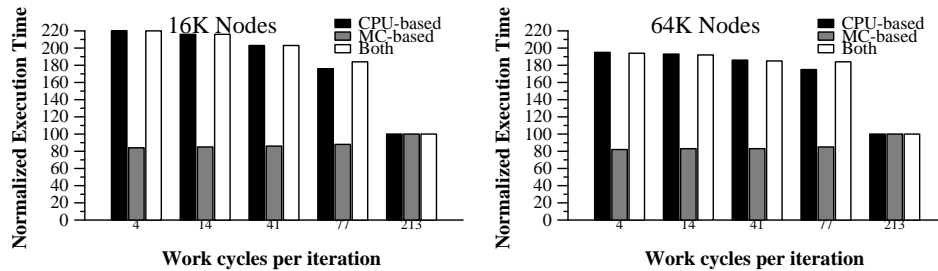


Figure 6. Normalized execution time for ternary tree.

`ptr>`, it can issue a load immediately. In this case, the number of memory cycles that can be hidden depends on the number of cycles required to execute all instructions before instruction 7 in the instruction window. In the microbenchmark, more than half of the instruction win-

dow contains floating-point instructions. A sequence of 32 or more floating-point instructions easily takes more than the average memory latency (e.g., division takes 12 cycles, with a 14-cycle repeat rate). The data for the next iteration will be ready before the current iteration

ends. As a result, there is not much room for improvement from pointer prefetching. The numbers in Figure 3 show that neither approach yields significant improvements on large iterations.

An addition/subtraction or a logical operation finishes in a couple of cycles, so a sequence of 64 of these operations is unlikely to exceed the memory latency. Prefetching should be useful for such integer instruction streams. Figure 4(b) shows results for a large loop consisting of hundreds of integer additions/subtractions and logical operations. For these linked-list experiments, all three prefetching approaches deliver good speedups. CPU-based prefetching can hide more latency than MC-based prefetching, which cannot hide delays on the system bus.

Figure 5 and Figure 6 show performance results for traversing a binary or ternary tree. For the biggest iterations, the total latency of three overlapped memory accesses is still less than the number of cycles spent working on each node. The memory latency can be completely hidden, and neither approach has an impact. CPU-based greedy prefetching and the combined approach dramatically slows down small iterations. The execution time of a small iteration is now essentially the total memory latency of two (for binary tree) or three (for ternary tree) cache misses. In contrast, MC-based prefetching delivers consistent speedups for small iterations. The performance of 16K-node lists is strongly affected by cache pollution. For the ternary tree, two-thirds of the CPU's prefetches are useless. Consequently, it suffers most from cache pollution due to prefetching.

## 6 Conclusions and Future Work

We have presented initial results for a greedy, memory-controller-based pointer prefetching technique: whenever the memory controller sees a request for a node in a linked data structure, it prefetches all objects directly pointed to by the this node. Our experiments use a modern, superscalar processor similar to the MIPS R10000 to evaluate its performance and to compare it with a comparable software approach.

Our results show that MC-based prefetching achieves a consistent 20% improvement for linked data structures accessed within medium to short loop iterations, outperforming the CPU-based software approach. Our experiments also reveal that the superscalar processor can successfully hide memory latency for linked data structures accessed within large loop iterations, which leaves little room for performance improvement from pointer prefetching.

Future work will further investigate MC-based pointer prefetching schemes, comparing them with

more sophisticated, CPU-based pointer prefetching algorithms such as jump pointers and prefetch arrays. These studies will expand our benchmark suite to include pointer-intensive, whole-program benchmarks (such as those containing a large number of short lists). We will explore the potential of software-directed MC-based prefetching, in which the processor controls when data is loaded into the MCache. Such an approach seems particularly promising, as it not only avoids the negative impacts of normal CPU prefetching on system bus and cache resources, but it effectively reduces the memory latency.

## References

- [1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings of the Second Annual Symposium on High Performance Computer Architecture*, pages 254–263, Feb. 1996.
- [2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, Nov. 1991.
- [3] J. Brooks. Single PE optimization techniques for the cray T3D system. In *Proceedings of the 1st European T3D Workshop*, Sept. 1995.
- [4] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.
- [5] K. Chan, C. Hay, J. Keller, G. Kurpanek, F. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, 47(1):25–33, Feb. 1996.
- [6] T.-F. Chen. Effective programmable prefetch engine for on-chip caches. In *Proceedings of IEEE/ACM 28th International Symposium on Microarchitecture*, pages 237–242, Nov. 1995.
- [7] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, Sept. 1993.
- [8] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233, May 1999.
- [9] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.
- [10] J. Edmondson, et al. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

- [11] Z. Fang, L. Zhang, J. Carter, S. McKee, and W. Hsieh. A comparison of online superpage promotion policies. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '00)*, June 2000.
- [12] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, June 1997.
- [13] J. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–65, Toronto, Canada, May 1991.
- [14] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.
- [15] G. Kane. *PA-RISC 2.0 Architecture*, 1996.
- [16] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the Sixth Annual Symposium on High Performance Computer Architecture*, pages 206–217, January 2000.
- [17] K. Lee. The NAS860 library user’s manual. Technical Report NAS Technical Report RND-93-003, NASA Ames Research Center, Mar. 1993.
- [18] C.-K. Luk and T. C. Mowry. Compiled-based prefetching for recursive data structure. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.
- [19] S. McKee, et al. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.
- [20] L. Meadows, S. Nakamoto, and V. Schuster. A vectorizing software pipelining compiler for LIW and super-scalar architectures. In *RISC'92*, pages 331–343, 1992.
- [21] S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 133–140, May 1996.
- [22] MIPS Technologies Inc. *MIPS R10000 Microprocessor User’s Manual, Version 2.0*, Dec. 1996.
- [23] T. Mowry, M. S. Lam, , and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.
- [24] V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim reference manual, version 1.0. Technical Report 9705, Rice University, 1997. available from <http://www-eece.rice.edu/rsim>.
- [25] S. Palacharla and R. Kessler. Code restructuring to exploit page mode and read-ahead features of the cray T3D. Technical Report Internal Report, Cray Research, Feb. 1995.
- [26] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structure. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.
- [27] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.
- [28] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
- [29] I. Sklenar. Prefetch unit for vector operation on scalar computers. *Computer Architecture News*, 20(4):31–37, Sept. 1992.
- [30] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, June 1998.
- [31] W. Wong and J.-L. Baer. DRAM on-chip caching. Technical Report UW-CSE-97-03-04, University of Washington Dept. of Computer Science and Engineering, Mar. 1997.
- [32] L. Zhang. URSIM reference manual. Technical Report UUCS-00-03, University of Utah, January 2000.
- [33] L. Zhang, J. Carter, W. Hsieh, and S. McKee. Memory system support for image processing. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.