Tolerating Memory Latency through Push Prefetching for Pointer-Intensive Applications

CHIA-LIN YANG
National Taiwan University
ALVIN R. LEBECK
Duke University
and
HUNG-WEI TSENG and CHIEN-HAO LEE
National Taiwan University

Prefetching is often used to overlap memory latency with computation for array-based applications. However, prefetching for pointer-intensive applications remains a challenge because of the irregular memory access pattern and pointer-chasing problem. In this paper, we proposed a cooperative hardware/software prefetching framework, the push architecture, which is designed specifically for linked data structures. The push architecture exploits program structure for future address generation instead of relying on past address history. It identifies the load instructions that traverse a LDS and uses a prefetch engine to execute them ahead of the CPU execution. This allows the prefetch engine to successfully generate future addresses. To overcome the serial nature of LDS address generation, the push architecture employs a novel data movement model. It attaches the prefetch engine to each level of the memory hierarchy and *pushes*, rather than *pulls*, data to the CPU. This push model decouples the pointer dereference from the transfer of the current node up to the processor. Thus a series of pointer dereferences becomes a pipelined process rather than a serial process. Simulation results show that the push architecture can reduce up to 100% of memory stall time on a suite of pointer-intensive applications, reducing overall execution time by an average 15%.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Prefetch, linked data structures, pointer-chasing, memory hierarchy

This research was supported in part by the ROC NSC Grant NSC-93-2752-E-002-008-PAE, DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637, CDA-95-12356, and EIA-99-72879, Duke University, Intel Corporation through a Ph.D. Fellowship and the T4E 2000 program.
Author's addresses: C.-L. Yang, H.-W. Tseng, and C.-H. Lee, Department of Computer Science and Information Engineering, National Taiwan University. No. 1, Sec. 4, Roosevelt Rd., Taipei, Taiwan, R.O.C.; email: yangcacsie.ntu.edu.tw; Alvin R. Lebeck, Department of Computer Science, Duke University, Durham, NC 27708.

1. INTRODUCTION

Since 1980, microprocessor performance has improved 60% per year, however, memory access time has improved only 10% per year. As the performance gap between processors and memory continues to grow, techniques to reduce the effect of this disparity are essential to build a high performance computer system. The use of caches between the CPU and main memory is recognized as an effective method to bridge this gap. If programs exhibit good temporal and spatial locality, the majority of memory requests can be satisfied by caches without having to access main memory. However, a cache's effectiveness can be limited for programs with poor locality.

Prefetching is a common technique that attempts to hide memory latency by issuing memory requests earlier than demand fetching a cache block when a miss occurs. Prefetching works very well for programs with regular access patterns. Unfortunately, many applications create sophisticated data structures using pointers, and do not exhibit sufficient regularity for conventional prefetch techniques to exploit. Furthermore, prefetching pointer-intensive data structures can be limited by the serial nature of pointer dereferences—known as the pointer chasing problem—since the address of the next node is not known until the contents of the current node is accessible.

For these applications, performance may improve if lower levels of the memory hierarchy could dereference pointers and proactively *push* data up to the processor rather than requiring the processor to *fetch* each node before initiating the next request. An oracle could accomplish this by knowing the layout and traversal order of the pointer-based data structure, and thus overlap the transfer of consecutively accessed nodes up the memory hierarchy. The challenge is to develop a realizable memory hierarchy that can obtain these benefits.

This paper presents a novel memory architecture to perform pointer dereferences in the lower levels of the memory hierarchy and push data up to the CPU. In this architecture, if the cache block containing the data corresponding to pointer p (*p) is found at a particular level of the memory hierarchy, the request for the next element (*p+offset) is issued from that level. This allows the access for the next node to overlap with the transfer of the previous element. In this way, a series of pointer dereferences becomes a pipelined process rather than a serial process.

The push architecture is a cooperative software/hardware prefetching framework that supports the above push model. The push architecture identifies instructions that traverse LDS (LDS traversal kernels) statically and uses a microcontroller to execute traversal kernels only. This microcontroller is called a prefetch engine (PFE). A PFE can successfully generate future addresses when it runs ahead of the CPU. To realize the push model, the push architecture attaches a PFE to each level of the memory hierarchy. These PFEs at different levels of the memory hierarchy execute LDS traversal kernels cooperatively. Cache blocks accessed by the PFEs at the L2 and memory levels are proactively pushed up to the CPU. In our earlier work [Yang and Lebeck 2000], the PFE is a specialized engine which only supports linked-list traversal. In this paper, the PFE is a programmable processor which can support a multitude of LDS traversal kernels. In addition, we address three main design issues in implementing the proposed push architecture:

(1) *Interaction among PFEs*. The PFEs at different levels of the memory hierarchy work cooperatively to perform the prefetching task. Therefore, the push architecture needs to specify how they interact with one another. In this paper, we establish a general interaction scheme among three PFEs, particularly how a PFE suspends and resumes execution and when the lower level PFEs should be activated.
(2) *Reducing the effect of redundant prefetches*. One potential problem of the push model is that the cache blocks pushed up by the lower level PFEs could already exist in the upper level. These are called redundant prefetches. Redundant prefetches can impact performance by wasting bus bandwidth or causing the PFE to run behind the main computation. To reduce the effect of redundant prefetches, the push architecture uses small data caches in the L2 and memory PFEs to capture recently accessed cache blocks.
(3) *Synchronization between the processor and PFEs*. Timing is an important factor for the success of a prefetching scheme. Prefetching too late results in useless prefetches; prefetching too early can cause cache pollution. In this paper, we present a throttle mechanism to control the prefetch distance according to a program's runtime behavior. The PFEs aggressively issue prefetch requests and suspend execution when the CPU is not able to keep up with the prefetching thread.

Simulation results show that the push model proposed in this paper is very effective at reducing the impact of the pointer-chasing problem on prefetching performance. For applications that have little computation between node accesses, the push architecture reduces execution time by 13% to 23% (using a single-issue, inorder programmable PFE), while the traditional pull method is not able to achieve any speedup. Increasing the issue width of the programmable PFE can further improve performance. With throttling, the push model can achieve performance comparable to a perfect memory system when there is enough computation between node accesses. Simulation results also show that adding a small data cache in the L2 and memory PFEs yields between 4% to 25% performance improvement for applications that experience a significant amount of redundant prefetches. We also examine if we can reduce the hardware cost of the push architecture by using fewer PFEs. Simulation results show that the push architecture using the PFE only at the L1 and main memory levels can achieve performance close to the one attaching the PFE to each level of the memory hierarchy. If the CPU is a multithreading processor, we can further reduce the hardware cost by invoking a helper thread to execute the LDS traversal kernel instead of using a PFE at the L1 level. We also compare the push architecture against jump-pointer prefetching [Roth and Sohi 1999], which is a pull-based prefetching mechanism but overcomes the pointer-chasing problem by providing additional pointers to connect nonadjacent nodes. Ideally, jump-pointer prefetching can tolerate any amount of memory latency as long as the jump-pointer interval is set appropriately. However, due to the limitations

```
list = head;                  while (list)
while (list)                  {
{
   p = list->patient;            p = list->patient;
   data = p->data;               data = p->data;
   process(data);                list = list->next;
   list = list ->next ;
}                             }
        (a)                             (b)
```

Fig. 1. Traversal kernel.

imposed by jump-pointer creation, jump-pointer prefetching shows little performance advantage or even adverse effects on some applications studied in this paper. Therefore, the push architecture outperforms jump-pointer prefetching for these applications. Simulation results show that the push architecture provides comparable average performance with jump-pointer prefetching.

The rest of this paper is organized as follows. Section 2 presents the high-level overview of the push architecture. The implementation details are addressed in Section 3. Section 4 presents our simulation results. Related work is discussed in Section 5. We conclude in Section 6.

2. THE PUSH ARCHITECTURE

The push architecture is a cooperative software/hardware prefetching framework designed specifically for linked data structures. Prefetching linked data structures are challenging because of address irregularity and the pointer-chasing problem. This section describes how the proposed push architecture overcomes these two challenges.

2.1 LDS Traversal Kernels

Traditional prefetching schemes rely on the regularity of the address stream to predict future addresses. These schemes work well for array-based numeric applications because the address stream presents arithmetic regularity. But they are not effective for linked data structures (LDS), because elements are allocated dynamically from the heap and adjacent elements are not necessarily contiguous in memory. Although linear layout of an LDS may be achieved through careful data placement [Chilimbi et al. 1998], the linearity property often disappears as the data structures evolve.

Instead of relying on address regularity, program structure can be used for future address generation [Roth et al. 1998]. Consider the LDS traversal example of a list traversal shown in Figure 1(a). The first instruction (list=head) loads the root address of the LDS list, while the three pointer loads form the body of the traversal kernel (see Figure 1(b)). The first load—called a *traversal load*—traverses internal links of the list. The second load—called a *data load*—accesses data other than memory addresses. The third load (list=list→next)—called a *recurrent load*—generates the address of the next element of the list structure.

By executing only the instructions in a traversal kernel, a prefetch engine could run ahead of the main computation, correctly generating future addresses. Roth et al. [1998] propose identifying traversal kernels dynamically in hardware. Their method represents the traversal kernel as a collection of explicit data dependence relationships. Once a pointer load is resolved, all the loads that depend on it are issued. This might be efficient for list-based applications but not necessarily for tree traversals, because it prefetches all child nodes once the parent node is obtained. This may result in many early prefetches if the program traverses the tree in depth first order.

In this work, we construct the LDS traversal kernel statically, so we can include information on traversal order. Although we construct kernels by hand in this paper, compiler or executable editor static analysis generally has larger analysis scope, and can consider traversal orders when constructing the kernels. However, there are cases when the traversal order or LDS access footprints (i.e., which fields of a LDS structure are accessed) cannot be determined statically.

In some cases, runtime information is available prior to traversal kernel execution and can be incorporated into the traversal kernel, for example, a key value used in searching a hash table. For access patterns with unpredictable traversal order, we can either aggressively prefetch all LDS elements or conservatively prefetch only those guaranteed to be accessed. The key thing is that the PFE does not perform computation beyond simple value comparison for flow control. Similarly, for applications that dynamically change the data structure during traversal, we can conservatively limit the prefetch distance to a few LDS elements. This prevents the PFE from prefetching down the wrong path if the CPU has modified the structure.

To support these various access patterns, we simply create different traversal kernels. Some run until they encounter a NULL pointer, others can include runtime information provided by the main CPU to determine which portion of a data structure to prefetch. Finally, we can limit prefetch distances by setting loop iteration counts or inlining a fixed number of instructions to control how many LDS elements are prefetched.

2.2 The Push Model

The conventional data movement model initiates all memory requests (demand fetch or prefetch) by the processor or upper level of the memory hierarchy. We call it the pull model because cache blocks are moved up the memory hierarchy in response to requests issued from the upper levels. Since pointer dereferences are required to generate addresses for successive prefetch requests (recurrent loads), these techniques serialize the prefetch process.

Consider the prefetch schedule shown in Figure 2(a), assuming each recurrent load is a L2 cache miss. The memory latency is divided into six parts ($r1$: sending a request from L1 to L2; $a1$: L2 access; $r2$: sending a request from L2 to main memory; $a2$: memory access; $x2$: transferring a cache block back to L2; $x1$: transferring a cache block back to L1). Using this timing, node $i+1$ arrives at the CPU $(r1+a1+r2+a2+x2+x1)$ cycles (the round-trip memory latency) after node i.

In contrast, the push model performs pointer dereferences at the lower levels of the memory hierarchy by executing traversal kernels in microcontrollers

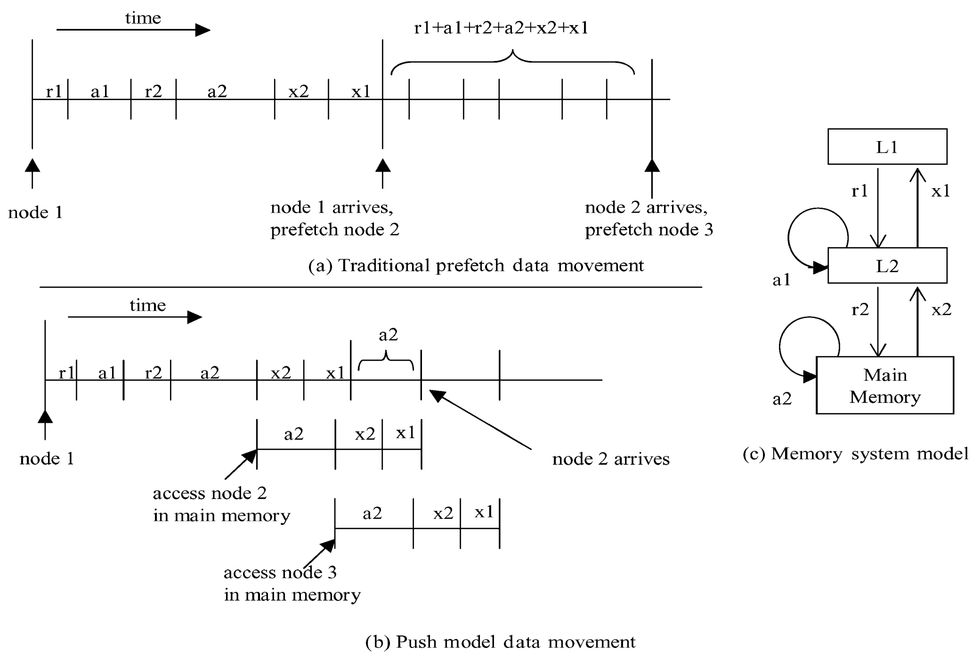

time
r1+a1+r2+a2+x2+x1
r1
a1
r2
a2
x2
x1
node 1
node 1 arrives, prefetch node 2
node 2 arrives, prefetch node 3
(a) Traditional prefetch data movement
time
a2
node 1
access node 2 in main memory
access node 3 in main memory
node 2 arrives
(b) Push model data movement
L1
L2
Main Memory
(c) Memory system model

Fig. 2. Data movement for linked data structures.

associated with each memory hierarchy level. The accessed cache blocks are proactively pushed up to the processor. This eliminates the request from the upper levels to the lower level, and enables overlapping the data transfer of node i with the RAM access of node $i + 1$. For example, assume that the request for node i accesses main memory at time t. The result is known to the memory controller at time $t + a2$, where $a2$ is the memory access latency. The controller then issues the request for node 2 to main memory at time $t + a2$, thus overlapping this access with the transfer of node 1 back to the CPU.

This process is shown in Figure 2(b) (Note that in this simple model, we ignore the possible overhead for address translation. Section 4 examines this issue). In this way, we are able to request subsequent nodes in the LDS much earlier than a traditional system. In the push architecture, node $i + 1$ arrives at the CPU $a2$ cycles after node i. From the CPU standpoint, the latency between node accesses is reduced to $a2$. From this analysis, we know that the DRAM access time and round-trip memory latency are two main architectural parameters that determine the relative performance of the push and pull models. In the Alpha 21264 architecture [Kessler 1999], the DRAM access time is 48 cycles and round-trip memory latency is 84 cycles. So the total latency is potentially reduced by 43%.[1]

[1]If the underlying memory system implements critical-word-first fetch, which transfers the requested word to the CPU before the rest of the cache block, the $x1$ and $x2$ components of the round-trip latency become 1 bus cycle. That effectively reduces the round-trip memory latency of the Alpha 21264 architecture to 68 cycles. Therefore, the push model could potentially reduce the memory latency by 29% instead of 43%.

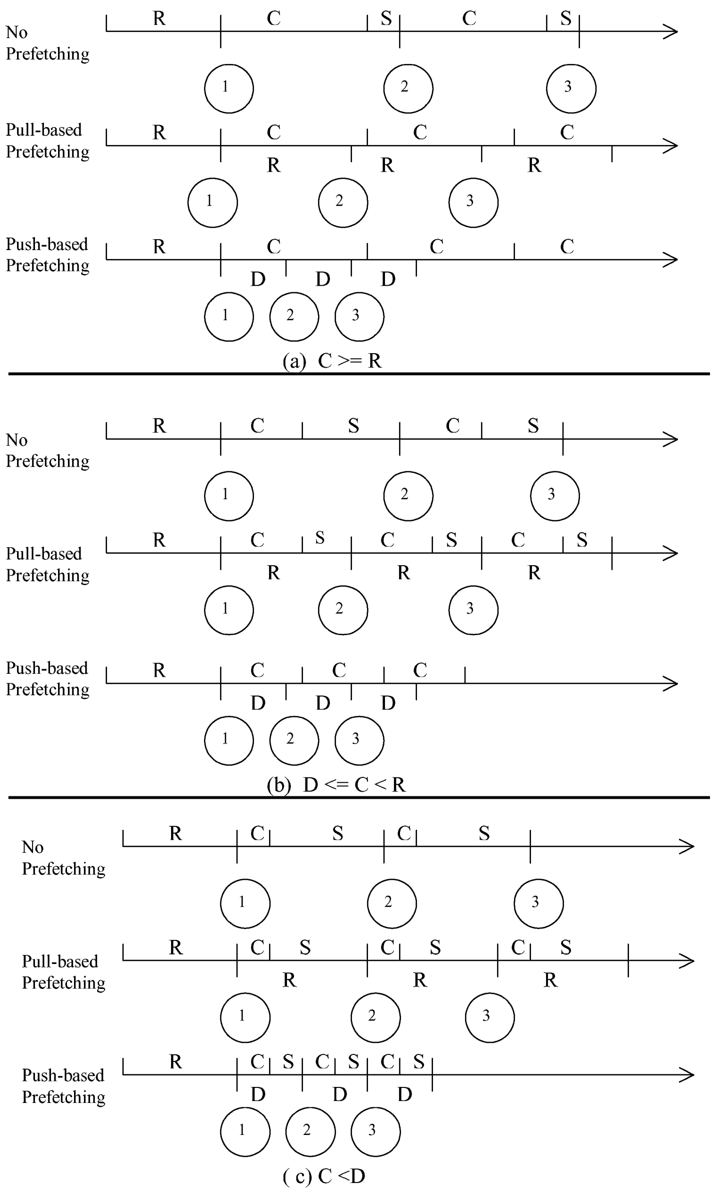


Fig. 3. Prefetching performance: push versus pull. R = round-trip memory latency; D = DRAM access time; C = computation time; S = memory stall time.

Besides the DRAM access time ($a2$) and round-trip memory latency ($r1 + a1 + r2 + a2 + x1 + x2$), the amount of computation (C) performed between node accesses in the LDS traversal is also an important factor that affects the relative performance of the push and pull model. Figure 3 illustrates how C affects prefetching performance for both the push and pull model in the following three scenarios: (1) $C \geq (r1 + a1 + r2 + a2 + x1 + x2)$

(2) $a2 \leq C < (r1 + a1 + r2 + a2 + x1 + x2)$, and (3) $C < a2$. Below we discuss each of these in more detail.

(1) $C \geq (r1 + a1 + r2 + a2 + x1 + x2)$. Figure 3(a) shows the scenario where C is greater than the round-trip memory latency. Without prefetching, an out-of-order processor is able to overlap memory latency with computation through dynamic instruction scheduling. However, it may not be able to completely hide memory latency because the limitation of the instruction look ahead window size. With prefetching, the PFE requests a subsequent node in the LDS once its address becomes available. Therefore, LDS nodes arrive at the CPU every $(r1 + a1 + r2 + a2 + a1 + x1 + x2)$ cycles for the pull model, and $a2$ cycles for the push model. Since C is greater than $(r1 + a1 + r2 + a2 + a1 + x1 + x2)$, both the push and pull model are able to bring data to the CPU in time, thus eliminating entire memory stall time. Therefore, pull-based prefetching performs as well as push-based prefetching.
(2) $a2 \leq C < (r1 + a1 + r2 + a2 + x1 + x2)$. Figure 3(b) shows the scenario where C is smaller than the round-trip memory latency but greater than the DRAM access time. In this case, the pull model only reduces a portion of memory stall time, while the push model can still bring data to the CPU in time, thus achieving performance close to a perfect memory system.
(3) $C < a2$. Figure 3(c) shows the scenario where C is smaller than the DRAM access time. In this case, neither the push or pull model can bring data to the CPU in time. The relative performance of the push and pull model is determined by the ratio between the DRAM access time and round-trip memory latency ($\frac{a2}{r1+a1+r2+a2+x1+x2}$). The smaller the ratio is, the larger performance improvement the push model can achieve over the pull model.

From the above analysis, we know that the push model outperforms the pull model when the amount of computation performed between node accesses is smaller than the round-trip memory latency. The actual performance advantage of push-based prefetching over pull-based for real applications also depends on (1) the contribution of LDS accesses to total memory latency and (2) the percentage of LDS misses that are L2 misses. We evaluate the push model performance using a set of pointer-based applications in Section 4.

To realize the push model, the push architecture attaches a prefetch engine to each level of the memory hierarchy as shown in Figure 4. Cache blocks accessed by the prefetch engine in the L2 or main memory level are pushed up to the CPU and stored either in the L1 cache or a prefetch buffer. The prefetch buffer is a small, fully associative cache, which can be accessed in parallel with the L1 cache. In the next section, we present the implementation details of the push architecture, including the architecture of the PFE, the interaction scheme among the PFEs, mechanisms to avoid early and redundant prefetches, and modifications to the cache/memory controller to support the push model.

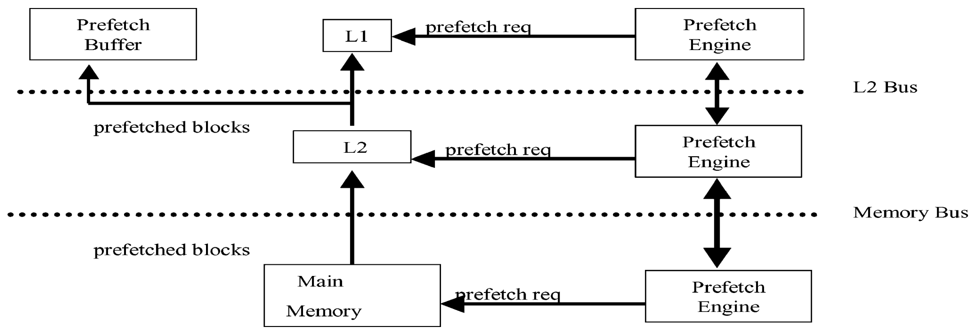


Fig. 4. The push architecture.

3. IMPLEMENTATIONS OF THE PUSH ARCHITECTURE

In this section, we describe the implementation details of the push architecture. Section 3.1 describes the architecture of the prefetch engine. Section 3.2 explains how the PFEs at different levels of the memory hierarchy interact with one another. Then, we present a mechanism to synchronize the CPU and PFE execution in Section 3.3. The push model also places new demands on the memory hierarchy. Section 3.4 addresses this issue. In Section 3.5, we present two variations of the push architecture. Finally, we discuss the issues on supporting the push architecture in the multiprogramming and multiprocessor systems in Section 3.6.

3.1 PFE Design

The task of the PFE is to execute the traversal kernel. It does not perform massive computation except for calculating addresses and value comparison for control flow. Therefore, the PFE does not need to have the full computation capabilities of a general-purpose processor, such as multiply, division, and floating-point operations. However, to execute different traversal patterns, the engine needs the ability to decode a sequence of instructions. We implement the prefetch engine as a 5-stage pipeline, in-order single issue processor. The processor pipeline stages are (1) *Fetch*: fetch instructions from the program instruction stream, (2) *Decode*: decode instructions, (3) *Execute*: execute ready instructions if the required functional units are available, (4) *Memory*: issue request to the memory system, and (5) *Writeback*: write results to the register file.

The PFE supports three types of load instructions. They are *ld_recurrent*, *ld_data*, and *ld_local*. *Ld_recurrent* and *ld_data* are used for recurrent and data loads, respectively. The use of *ld_recurrent* is explained in the next section. The *ld_data* instruction is a nonbinding load similar to the existing prefetch instructions. It is used for loads that do not have dependent instructions. The third kind is *ld_local*, which is used to access local memory inside an engine. The local memory serves as a stack for recursive traversal kernels and can be accessed in one cycle. The size of the stack needed is proportional to the height of a tree. Thus, the maximum stack size for a 32-level tree is 4K, assuming all

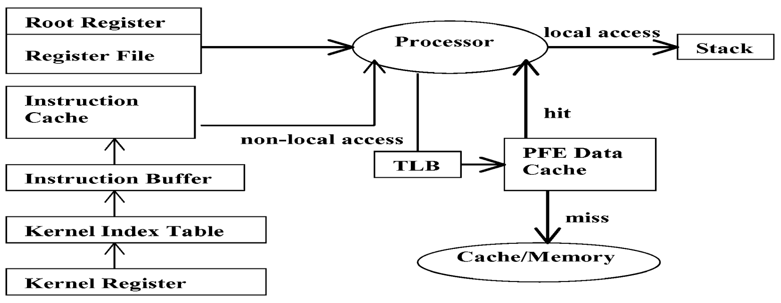


Fig. 5. The prefetch engine.

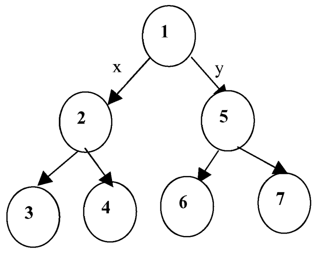
registers (32 registers) are modified in the program. So one cycle access time is feasible.

The engine structure is shown in Figure 5. The PFE contains an instruction buffer to store traversal kernels. Traversal kernels are downloaded to the PFE through a memory-mapped interface prior to program execution. Since a program can have multiple traversal kernels, each kernel is associated with an identifier. The kernel identifier is passed to the PFE at run time and stored in the kernel register. The kernel index table stores the information for locating a traversal kernel in the instruction buffer using its kernel identifier. When the PFE detects a write to the kernel register, it searches the kernel index table using this register value and loads the corresponding traversal kernel to the instruction cache. The PFE is activated when the root address of the LDS being traversed is written into the root register. The traversal kernel may need other run time information in addition to the root address, such as a key value used in searching a hash table. So a portion of PFE registers are memory mapped to allow the CPU to convey other run time information down the PFE if necessary.

The PFE contains a TLB for address translation for nonlocal loads because the cache/memory is indexed using physical addresses. We assume hardware-managed TLB. TLBs at different levels of the memory hierarchy are updated independently. The L2 and memory PFE contains a local data cache to reduce the performance impact of redundant prefetches, which push up cache blocks that already exist in the upper level of memory hierarchy. Redundant prefetches could cause the PFEs to run behind of the main execution because memory accesses are satisfied more slowly in the lower levels of the memory hierarchy. This is particularly important for applications with small amounts of locality, such as depth-first tree traversal. When the execution moves up the tree, all the parent nodes are revisited. For nodes close to the leaves, the second access could be a cache hit and become a redundant prefetch. Therefore, we use a data cache in the L2 and memory PFE to capture this locality. Only memory requests that miss in the local data cache are sent to the cache/memory controller. In this way, we can avoid issuing prefetches for those cache blocks that have been pushed up to the upper level of the memory hierarchy due to earlier prefetches. Therefore, the size of the local data cache should be the same as the size of the push buffer at the L1 level.

```
    00400950 addiu $sp[29],$sp[29],-56
    00400958 sw $ra[31],48($sp[29])
    00400960 sw $s8[30],44($sp[29])
    00400968 sw $s0[16],40($sp[29])
    00400970 addu $s8[30],$zero[0],$sp[29]
    00400978 addu $s0[16],$zero[0],$a0[4]
    00400980 beq $s0[16],$zero[0],004009a8
(x) 00400988 lw $a0[4],4($s0[16])
    00400990 jal 00400950 <K_TreeAdd>
(y) 00400998 lw $a0[4],8($s0[16])
    004009a0 jal 00400950 <K_TreeAdd>
    004009a8 addu $sp[29],$zero[0],$s8[30]
    004009b0 lw $ra[31],48($sp[29])
    004009b8 lw $s8[30],44($sp[29])
    004009c0 lw $s0[16],40($sp[29])
    004009c8 addiu $sp[29],$sp[29],56
    004009d0 jr $ra[31]
```

(a) Traversal Kernel



(b) Data Structure

Recurrent Load PC	Resume PC
400988	400998
400998	4009a8

(c) Resume PC Table

Fig. 6. Tree traversal.

3.2 Interaction Among Prefetch Engines

The PFEs need two pieces of information to start execution: the root address and the kernel identifier. These values are communicated to the engines through store operations to memory-mapped registers of the PFEs. The root address is only written to the L1 engine, which then triggers the L1 PFE.

The idea behind the push model is to perform pointer dereferences at the level where the LDS element resides such that the cache blocks can arrive at the CPU earlier than a pull-based prefetch. Since a recurrent load is responsible for loading the next node address, we use it as a hint for which memory level should perform prefetching activity. The ld_recurrent instruction is used to distinguish a recurrent load from other loads. When a recurrent load causes a L1 miss, the cache controller signals the L1 PFE to stop execution. If this recurrent load is a hit in the L2 cache, the loaded value is written to the root register of the L2 PFE, which triggers the engine. Otherwise, on a L2 miss, the memory PFE begins prefetching.

An important aspect of the push architecture is the interaction among the PFEs. Consider traversing a binary tree in depth-first order, as shown in Figure 6. The assembly code is the traversal kernel. Assume that the recurrent load x (in Figure 6(a)), between node 1 and 2 in Figure 6(b), misses in both the L1 and L2 cache. The L1 PFE suspends execution, and the memory engine starts prefetching. However, the memory PFE has only enough information to prefetch the subtree A (nodes 2, 3, and 4). The L1 engine should resume execution at point y after the memory PFE finishes prefetching subtree A. To achieve this, we need to answer the following two questions:

1. What should the L1 engine do when a recurrent miss occurs so it can correctly resume execution?
2. When the memory engine finishes prefetching the subtree A, how does it notify the L1 engine to resume execution?

When the L1 cache controller detects that the prefetch recurrent load x (which corresponds to instruction 0x400988) misses in the L1, the L1 PFE stops execution. To resume execution at point y correctly, the PFE needs to set its current program counter (PC) to 0x400998, which is the recurrent load for the other child (node 5). By the time the PFE is signaled to stop execution, it might have progressed to the next tree level and modified some of the registers. To resume at point y, we must ensure the register values are valid. To achieve this, the PFE needs two pieces of information: the program counter of the recurrent load x to determine the resume PC and the stack pointer value when x is issued, so it can correctly restore register values.

We annotate a PFE recurrent load with its PC (RPC) and stack pointer value (SP) when it is issued. When the cache controller signals the engine to stop execution, these two items are used to restore the correct state. First, the RPC is used to access a small table (see Figure 6(c)) to obtain the correct resume PC. This table is easily constructed statically, and is downloaded as part of the traversal kernel.

To restore the register values, we can take advantage of the stack maintained by the traversal kernel itself. Recursive programs always save the current register values on the stack before proceeding to the next level. Thus, we can restore registers from the stack with the SP information. The tricky part is that the engine can be in the middle of saving registers to the stack when it is signaled to stop. In our example shown in Figure 6, the code segment from 0x400958 to 0x400968 is responsible for saving register values to the stack. When the engine is signaled to stop execution due to the missing recurrent load x (which corresponds to instruction 0x400988), it can be executing any instruction between 0x400950 and 0x400978 (the PFE should stall at 0x400978 because of a dependency hazard).

If the PFE has finished executing the code sequence from 0x400958 to 0x400968, we can safely restore the register values from the stack. Otherwise, the current register values are valid and the engine should not restore registers from the stack. To determine whether to restore the register values from the stack or not, we use a register to store the last PC of the save codes in the kernel (0x400968 in this example). If the current PC is greater than this value at the time the engine is signaled to stop execution, the engine should restore registers from the stack and set the stack pointer to SP. Otherwise, the engine only needs to set the stack pointer to SP. The other solution is to delay restoring registers from the stack until the PFE finishes saving registers. This is similar to the approach used in making interrupts to achieve atomic execution.

To resume execution we assume a dedicated bus, called the PFE channel, for communication between PFEs. From the previous example, when the memory engine finishes prefetching subtree A, it broadcasts a Resume message, and the L1 engine, which suspends execution because of a recurrent load miss, resumes execution when it observes this Resume message.

Note that the interaction scheme described above imposes limitations on the push architecture since it supports only tree and linked-list traversals. However, we believe that the supported traversal kernels are commonly seen in pointer-intensive applications, such as database retrieval and 3D graphics applications. If the traversal kernels do not fall into the supported access patterns, we can simply invoke only the engine on the memory side (i.e., the 1_PFE architecture described in Section 3.5) to take advantage of the proposed push model.

Having described the PFE structure and the interaction scheme among three level PFEs, we now present two architectural features for preventing redundant and early prefetches.

3.3 Synchronization Between the Processor and PFEs

Timing is an important factor for the success of a prefetching scheme. Prefetching too late results in useless prefetches; prefetching too early can cause cache pollution. The prefetch schedule depends on the amount of computation available to overlap with memory accesses, which can vary throughout application execution. This section presents a throttle mechanism that adjusts the prefetch distance according to the program's runtime behavior.

The idea of this scheme is to throttle PFE execution to match the rate of processor data consumption. Our approach is built around the prefetch buffer, where the PFEs produce its contents and the CPU consumes them. We augment each cache line of the prefetch buffer with a free bit [Smith 1982a]. The free bit is set to 0 when a new cache block is brought in. Once this block is accessed by the processor, the free bit is set to 1 and this cache block is moved into the L1 cache. The desirable behavior is a balance between the producing and consuming rate. If the prefetch buffer is full, it indicates there is a mismatch in these two rates and the following three scenarios are possible:

(1) The engine is running on the correct path but is too far ahead of the CPU,
(2) The engine is on the wrong path, and the PFEs place incorrect cache blocks in the buffer, or
(3) The engine is on the correct path but is behind the CPU execution.

When the prefetch buffer is full, the CPU broadcasts a Pb_Full message to the PFEs through the PFE channel, and the active engine suspends execution. In the first case, the CPU will eventually access the prefetch buffer. Once an entry is freed up, the CPU broadcasts a Pb_Free message, and the active engine resumes execution. Note that because of the delay for the communication, cache blocks can still arrive at the L1 level even though the prefetch buffer is full. The LRU policy is used in this case to replace blocks in the prefetch buffer. The replaced blocks are stored in the L1 cache for later accesses.

For the next two cases, most of the time, the CPU will never access data in the prefetch buffer, and the PFE should start at a new point. The method to detect these cases is to use a recurrent load as a synchronization point. If the prefetch buffer is full and a recurrent load is a miss in both the L1 cache and prefetch buffer, it is very likely that the engine is behind or running on the wrong path. Thus, the CPU broadcasts a message to tell the PFEs to stop prefetching. This recurrent load then serves as a new triggering point.

The complete PFE state diagram with throttle mechanism is shown in Figure 7. The PFE transits to state 1, the active state, when its root register is written. When a recurrent load miss occurs, the engine suspends execution, restores the correct state and moves to state 2. When a PFE finishes executing

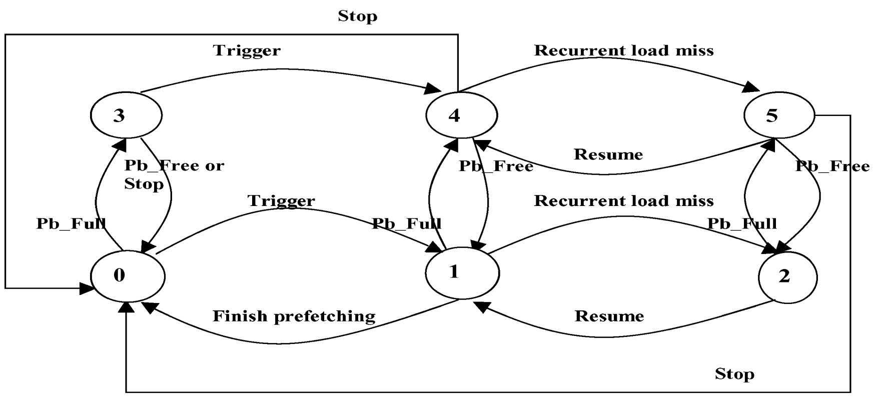


Fig. 7. Complete PFE state diagram.

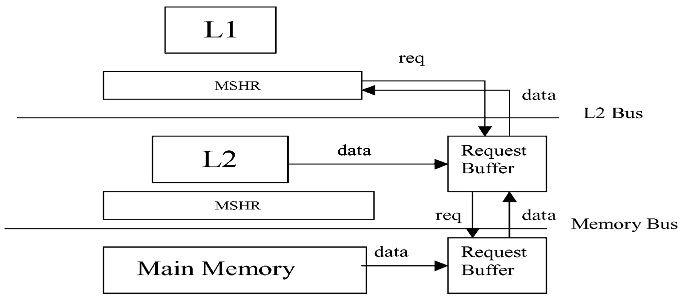


Fig. 8. The block diagram of the memory hierarchy.

the traversal kernel, it broadcasts a Resume message on the PFE channel and returns to state 0. A PFE in state 2 resumes execution when it observes a Resume message. When an engine observes a Pb_Full message, it moves to state 3, 4, or 5 depending on what the current state is. Note that a PFE can still receive Recurrent load miss, Resume and Trigger events after observing a Pb_Full message. So an engine at state 3, 4, and 5 also needs to respond to these events (i.e., transition among these three states). An engine at state 3, 4, or 5 moves to state 0, 1, or 2 when observing a Pb_Free message and returns to state 0 when observing a Stop message.

3.4 Modifications to the Cache/Memory Controller

The push model also places new demands on the memory hierarchy to correctly handle cache blocks that were not requested by the processor. This section addresses this issue.

Before presenting the required modification to the cache and memory controllers to support the push model, we first describe the implementation details of the memory hierarchy. The block diagram of the memory hierarchy is shown in Figure 8. The L1 and L2 cache controllers include miss information/status holding registers (MSHR) for each miss that will be handled concurrently [Kroft 1981]. When a cache miss occurs, a MSHR is allocated to store the information

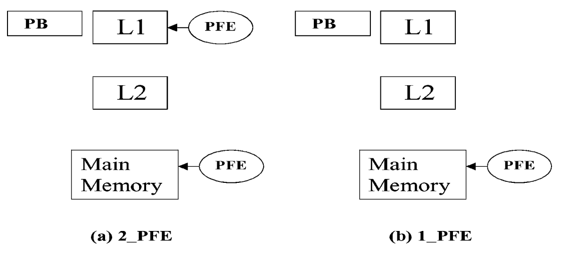


Fig. 9. Variations of the push architecture.

of this miss, such as its address. A memory request travels down the lower levels of the memory hierarchy along with its MSHR identifier. The L2 and memory levels of the memory hierarchy contain a small buffer that stores requests from the CPU, cache coherence transactions (e.g., snoop requests) and requested cache blocks. A requested cache block is transferred up the memory hierarchy along with its MSHR identifier.

In the push model, when a demand request arrives at the lower levels of the memory hierarchy, a prefetch request to the same cache block could already exist. So before serving any request, the controller checks if a request to the same cache block already exists in the request buffer. If such a request exists, the controller simply drops this demand fetch. A prefetched cache block is transferred up the memory hierarchy along with its address. When a prefetched cache block arrives at the upper level, its address is used to locate the matching MSHR, thus satisfying corresponding demand fetches. In this way, the memory latency of a demand fetch that is issued before prefetched data arrives can be partially hidden. The PFE might issue more than one request to one cache block. Since the controller always checks if a request to the same cache block exists in the request buffer before serving any request, only the first prefetch request accesses the cache/memory.

It is not guaranteed that a prefetched cache block will find a corresponding MSHR. One could exist if the processor executed a load for the corresponding cache block. If there is no matching MSHR, the returned cache block can be stored in a free MSHR or in the request buffer. If neither one is available, this cache block is dropped. Dropping these cache blocks does not affect the program correctness, since they are a result of a nonbinding prefetch operation, not a demand fetch from the processor. Before updating the cache using prefetched data, the cache controller needs to check if the corresponding cache block exists in the cache (redundant prefetch) to avoid keeping multiple copies of the same block in the cache.

3.5 Variations of the Push Architecture

The push architecture described above attaches the PFE to each level of the memory hierarchy (3_PFE). To reduce the hardware cost, we present two design alternatives: 2_PFE and 1_PFE. 2_PFE attaches the PFE to the L1 and main memory levels, while 1_PFE only uses one PFE at the main memory level.

Figure 9 shows the block diagram of these two architectures. 2_PFE performs pull-based prefetching between the L1 and L2 cache until a recurrent load misses in the L2 cache. As a result, data in the L2 cache is *pulled* up to the CPU instead of being *pushed* up as in 3_PFE. Based on the analytical model presented in Section 2, the latency between recurrent prefetches is $r1+a1+x1$ for the pull model and $a1$ for the push model for objects that exist in the L2 ($r1$: time to send a request from L1 to L2; $a1$: L2 access time; $x1$: time to transfer a cache block from L2 to L1). So the push model brings data to the L1 level only $r1 + x1$ cycles earlier than the pull model. For most computer systems, $r1 + x1$ is only a small portion of the round-trip memory latency. So 2_PFE should perform comparably to 3_PFE. We can further reduce the hardware cost of 2_PFE if the CPU is a multithreading processor. Instead of using a separate processor at the L1 level for prefetching, we can invoke a helper thread to execute the LDS traversal kernel. This approach has been used in several pre-execution studies [Roth and Sohi 2001; Zilles and Sohi 2001]. In this way, 2_PFE only needs to employ one PFE at the main level.

In the 1_PFE architecture, all prefetches are issued from the main memory level. 1_PFE greatly simplifies the push architecture design because the interaction issue among the PFEs no longer exists. 1_PFE should work effectively if a large portion of the LDS being traversed exists only in main memory. However, for applications in which the L2 cache is able to capture most of the L1 misses, load instructions are resolved more slowly in the memory PFE than in the CPU. So 1_PFE may not be able to achieve any prefetching effect because the memory PFE is very likely to run behind the CPU.

3.6 Discussion

in this section, we discuss issues on supporting the push model in the multiprogramming and multiprocessor systems.

3.6.1 *Context Switch Issues.* In the multiprogramming environment, a process could suspend execution at any instance. The operating system is responsible for saving and restoring process state correctly during a context switch. In the push architecture, in addition to the CPU state, the operating system also needs to save and restore the state of three PFEs if we want the PFE to resume execution at the point when it is interrupted. This incurs extra overhead for a context switch. One design alternative is to reactivate the PFE using a recurrent load miss as a triggering point when a suspended process resumes execution. The traversal kernels for the running process can be downloaded into the PFE during a context switch. The other approach is to wait until the first instruction TLB miss occurs to download the traversal kernels. This requires the operating system to associate a process's traversal kernel information with its instruction TLB entries. This paper focuses on the performance impact of the push model on a single program, so we do not evaluate how a context switch affects prefetching performance further.

3.6.2 *Impacts on the Multiprocessor Memory Systems.* To support the push model on the multiprocessor memory systems, we can attach a PFE to each memory module as proposed in Hughes [2002]. A memory-side prefetch engine issues prefetch requests to its local memory. The requested data is pushed back

Table I. Parameters of the Simulated Architecture

Out-of-Order Core	5 stage, 4-way superscalar, out-of-order pipeline with 64 RUU entries and a 16 entry load-store queue
Branch Prediction	2-level branch predictor with a total of 8192 entries
Memory System	128-entry hardware-managed TLB with 30-cycle miss penalty 32 KB, 32B-lines, 2-way L1 D/I caches with 2 read/write ports and 1 cycle access latency 512 KB, 64B-lines 4-way unified L2 cache with 18 cycles round-trip latency Both the L1 and L2 caches are lock-up free caches with 8 outstanding misses Latency to main memory after a L2 miss is 66 cycles

to the requesting processor if it is found in the local memory. Since the entire data structures being traversed may distribute on different memory modules, multiple memory-side PFEs need to work cooperatively to finish the prefetching task. The interaction scheme among PFEs at different levels of memory hierarchy described in Section 3.2 can be extended to handle the interaction among the PFEs at different memory modules. We leave the detailed design as future work.

4. EVALUATION

In this section, we present simulation results that demonstrate the effectiveness of the proposed prefetching scheme. Section 4.1 describes our simulation framework. In Section 4.2, we show the analysis of a microbenchmark to provide insight into the performance advantage of the push model over the conventional pull model. We then examine the prefetching scheme in detail using a set of pointer-based applications in Section 4.3. We first evaluate the performance improvement achieved by the prefetching scheme using a single-issue processor as the PFE. We then analyze how the issue width of a PFE affects performance. We finish by evaluating the various push architectures discussed in Section 3.5.

4.1 Methodology

To evaluate the push model we modified the SimpleScalar simulator [Burger et al. 1996] to include our PFE implementation. SimpleScalar models a dynamically scheduled processor using a register update unit (RUU) and a load/store queue (LSQ) [Sohi 1990]. The processor pipeline stages are (1) *Fetch*: Fetch instructions from the program instruction stream, (2) *Dispatch*: Decode instructions, allocate RUU, LSQ entries, (3) *Issue/Execute*: Execute ready instructions if the required functional units are available, (4) *Writeback*: Supply results to dependent instructions, and (5) *Commit*: Commit results to the register file in program order, free RUU, and LSQ entries.

Our base model uses a 4-way superscalar, out-of-order processor with 64 RUU entries and a 16 entry load-store queue. The branch unit includes a 2-level branch predictor with a total of 8192 entries. The memory system consists of a 128-entry TLB, a 32 KB, 32-byte cache line, 2-way set-associative first-level data and instruction caches and a 512 KB, 64-byte cache line, 4-way set associative shared second-level cache. We implement a hardware-managed TLB with a 30 cycle miss penalty. Both the L1 and L2 caches are lockup-free caches that can have up to eight outstanding misses. The L1 cache has two read/write ports, and can be accessed in a single cycle. The second-level cache is pipelined with an 18 cycle round-trip latency. Latency to main memory after an L2 miss is 66 cycles. We derived the memory system parameters based on the Alpha 21264 design [Kessler 1999], which has a 800 MHz CPU and 60 ns DRAM access time (48 CPU cycles). So the ratio between the DRAM access time to the round-trip memory latency is 48 to 84.

We first evaluate the performance of the push model on the 3_PFE architecture (i.e., attaching the PFE to each level of the memory hierarchy). The PFE is an in-order, single-issue processor, which runs at the same clock rate as the CPU. We believe that this is a reasonable assumption. An embedded processor like StrongARM-2 [Leibson 2000] is able to achieve 800 MHz clock rate. We believe that even in the future, the PFE's clock rate is able to catch up general-purpose processors since the hardware complexity of the PFE is much simpler than a general-purpose processor. We then vary the PFE issue width and the number of PFEs used in the push architecture. We simulate a 32-entry fully associative prefetch buffer with four read/write ports, that can be accessed in parallel with the L1 data cache. Both the L2 and memory PFEs contain a 32-entry fully associative data cache. The memory PFE is close to the DRAM chips and can be incorporated into DRAM chips [Patterson et al. 1997] if the memory controller is not close by. Contention on the bus and cache/memory ports between prefetches and demand fetches is modeled. Demand fetches are always given priority over prefetches. Since we do not add additional cache/memory ports to support the push model, we assume that the cache and memory access latency remains the same as the base model. We place a 128-entry TLB in the L2 and memory PFEs. Since the L1 PFE is close to the CPU, it shares one TLB with the CPU. TLBs at different levels of the memory hierarchy are updated independently.

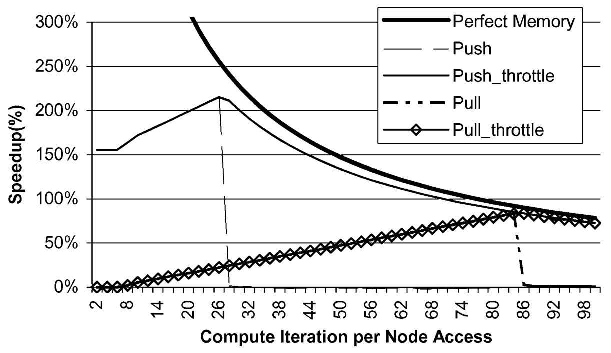
We simulate the pull model by using only a single PFE at the L1 cache. This engine executes the same traversal kernels used by the push architecture, but pulls data up to the processor. We use CProf [Lebeck and Wood 1994] to identify kernels that contribute the most cache misses, and construct only these kernels by hand. As mentioned in Section 2, we may not have enough information to determine the accurate traversal path when constructing kernels statically. In this case, the kernel is constructed to prefetch only those LDS elements that are guaranteed to be accessed.

4.2 Microbenchmark Results

We begin by examining the performance of both prefetching models (push and pull) using a microbenchmark. Our microbenchmark simply traverses a linked list of 32,768 32-byte nodes 10 times, see Figure 10(a). Therefore, all LDS loads are recurrent loads. We also include a small computation loop between each node access. This allows us to evaluate the effects of various ratios between computation and memory access. To eliminate branch effects, we use a perfect branch predictor for these microbenchmark simulations.

```
for (i = 0; i<10; i++)
   node = head;
   while (node)
      for (j=0; j < iters; j++)
         sum = sum+j;
      node = node->next;
   endwhile
endfor
```

(a) Microbenchmark source code



(b) Push vs. pull: microbenchmark results

Fig. 10. Microbenchmark.

Figure 10(b) shows the speedup for each prefetch model over the base system versus compute iterations on the x-axis. The thick solid line indicates the speedup of a perfect memory system where all references hit in the L1 cache. To examine the effect of the throttle mechanism, we tested the microbenchmark with four different configurations. Push_base and push_throttle are the push model without and with the throttle mechanism, while pull_base and pull_throttle correspond to the respective pull-based design.

We consider three distinct phases in Figure 10(b). Initially, the computation between node accesses is very short, such that the PFEs need to continue forging ahead to keep up with the CPU, thus push_base and push_throttle perform comparably. The push model achieves nearly 150% speedup even with the shortest computation loop. The pull model performs poorly in this phase, because the PFE cannot run far enough ahead of the CPU to produce useful prefetches.

As the computation per node access increases, the push model reaches its peak performance, close to the perfect memory system. But after this point, there is a sudden performance drop for push_base, because the PFE runs too far ahead of the CPU, and replaces blocks from the prefetch buffer. In contrast, push_throttle successfully throttles the PFE, and performs close to the perfect memory system. Also, the pull model starts improving performance, but it remains far below push_throttle. In the last phase, the pull model reaches its peak performance and throttling becomes important, with both models performing equally.

This experiment shows that the push model clearly has significant performance advantage over the pull model for applications with little computation between node accesses. The push model may perform equal to the pull model for applications with long computation between node accesses. However, for these applications, cache performance is a smaller component of overall execution time.

4.3 Macrobenchmark Results

While the above microbenchmark results provide some insight into the performance of the push and pull models, it is difficult to extend those results to

Table II. Benchmark Characteristics

Benchmark	Linked Data Structures	Input Parameters	Data Set Size	Traversal Kernel Size	L1/L2 Read Miss Rate
bh	Octree	4K bodies	720 KB	108B	3%/45%
bisort	Binary tree	250,000 numbers	1.5 MB	24B	6%/36%
em3d	Lists	2000 nodes, arity 10	1.6 MB	92B	57%/19%
health	List	5 levels, 500 iterations	925 KB	32B	26%/53%
mst	Array of lists	1024 nodes	20 KB	28B	16%/58%
perimeter	Quadtree	4kx4k image	6.4 MB	184B	9%/70%
treeadd	Binary tree	20 levels	32 MB	68B	12%/50%
tsp	Lists	100,000 cities	5.1 MB	48B	5%/48%
voronoi	Binary tree	60,000 points	11 MB	52B	2%/15%
rayshade	List	teapot data set	671 KB	68B	13%/88%
mcf	Arrays of pointers	Reference data sets	97 MB	1K	52%/70%

real applications. Variations in the type of computation and its execution on dynamically scheduled processors can make it very difficult to precisely predict memory latency contributions to overall execution time.

To address this issue, we use the Olden pointer-intensive benchmark suite [Roger et al. 1995]. Olden contains ten pointer-based applications written in C. It has been used in the past for studying the memory behavior of pointer-intensive applications [Luk and Mowry 1996; Roth et al. 1998; Roth and Sohi 1999]. In addition to the Olden benchmark, we also tested the push scheme on two general applications, Rayshade [Kolb] and mcf. Rayshade is a real-world graphics application that implements a raytracing algorithm. Mcf is taken from SpecINT2K suite. Table II lists the types of linked-data structures, the data set size, and traversal kernel size used in this study. Note that we only list the data structure types for the traversal kernels executed by the PFEs. We omit the power application because it has a low (1%) miss rate. We evaluate the push model performance for both a perfect and 128-entry TLB. Since the proposed push architecture is designed to improve data cache performance, by assuming a prefetch TLB, we are able to evaluate the effectiveness of the push architecture from memory stall time reduction. Using a realistic TLB configuration could decrease the overall execution time reduction achieved by the push architecture. However, for this set of benchmark tested, a 128-entry TLB has less than 1% impact on the push architecture performance except for health.[2] Therefore, we only present simulation results assuming a perfect TLB in this paper. Note that unlike the microbenchmark results, the branch prediction effect are considered when evaluating this set of macrobenchmarks.

4.3.1 *Performance Comparison Between The Push and Pull Model.* Figure 11 shows execution time normalized to the base system without prefetching. For each benchmark, the three bars correspond to the base, push and pull models, respectively. Execution time is divided into two components, memory stall time and computation time. We obtain the computation time by assuming

[2] 128-entry has 5% performance impact on health.

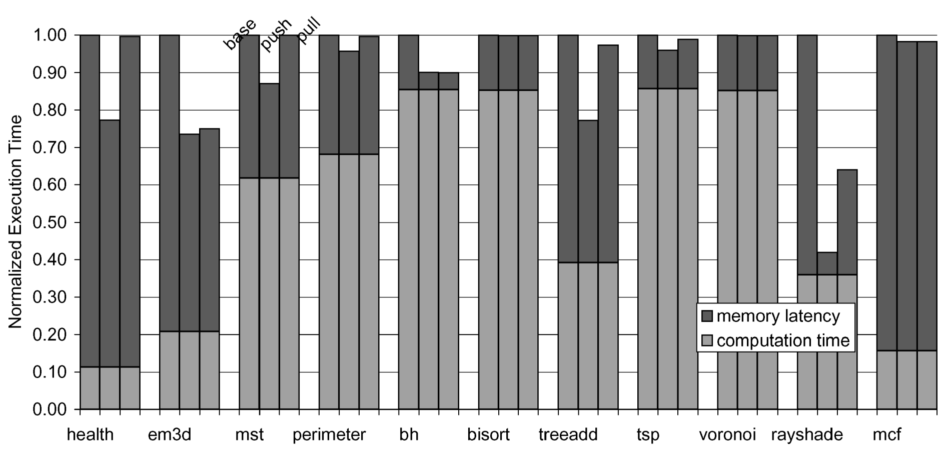


Fig. 11. Performance comparison between the push and pull model.

a perfect memory system. For the set of benchmarks with tight traversal loops (health, mst, treeadd), the push model is able to reduce between 25% and 38% of memory stall time (13% to 25% overall execution time reduction) while the pull model can only reduce the stall time by at most 4%. Perimeter traverses down a quad-tree in depth-first order, but has an unpredictable access pattern once it reaches a leaf. Therefore, we only prefetch the main traversal kernel. Although perimeter performs some computation down the leaves, it has very little computation to overlap with the memory access when traversing the internal nodes. So the pull model is not able to achieve any speedup, but the push model reduces the execution time by 4%.

For applications that have longer computation lengths between node accesses (bh, rayshade, em3d), we expect larger reductions in memory stall time than for programs with little computation between node accesses. From Figure 11, we see that the push model performs close to a perfect memory system for rayshade and bh (89% and 100% memory stall time reduction), reducing execution time by 57% and 36%, respectively. The pull model achieves similar improvements for bh, but reduces execution time by only 39% for rayshade. The exception is em3d, which shows only a 31% reduction in memory stall time and 25% in execution time, with similar performance for the pull model.

When the pull model performs comparable to the push model, we expect performance close to a perfect memory system (see Section 4.2). While this is true for bh, it is not for em3d. Em3d has poor L1 cache performance (57% load miss rate), but the L2 cache is able to capture 80% of these misses. For LDS elements that exist in the L2 cache, the latency between recurrent prefetches in the pull model is $r1 + a1 + x1$ ($r1$: time to send a request from L1 to L2; $a1$: L2 access time; $x1$: time to transfer a cache block from L2 to L1). For the push model, the latency between recurrent prefetches is $a1$ since prefetches are issued from the L2 level. So the push model brings data to the L1 level $r1 + x1$ cycles earlier than the pull model. $R1 + x1$ is equal to 6 in this experiment, so the push model does not show significant performance improvement over the pull model for em3d.

Bisort and tsp dynamically change the data structure while traversing it. As mentioned in Section 3, the prediction accuracy is low for this type of application. For tsp, we are able to identify some traversal kernels that do not change the structure dynamically. The push model is able to reduce the execution time by 4% and the pull model 1%. For bisort, neither the push or pull model is able to improve performance because the prediction accuracy is low (only 20% of prefetched cache blocks are useful). By only prefetching one node ahead, both the push and pull can reduce the execution time by 3%. For mcf, the data structure traversed in the main kernel is arrays of pointers. The traversal pattern is not supported by the interaction scheme among three PFEs described in Section 3.2. As a result, the PFE at the lower levels of the memory hierarchy is never triggered. Therefore, the push model performs the same as the pull model.[3] Voronoi uses pointers, but array and scalar loads cause most of the cache misses. So we are not able to see any performance improvement for either the push or pull model.

The macrobenchmark results match our expectations based on the microbenchmark analysis in the previous section. The push model is effective even when the applications have very tight loops where the performance of the traditional pull model is limited because of the pointer-chasing problem. For applications with enough computation between node accesses, the push model is able to achieve performance close to the perfect memory system when the pull model is still not able to deliver comparable performance.

The pull-based prefetching mechanism assumed in the above discussion does not employ techniques to overcome the pointer-chasing problem. Roth and Sohi [1999] propose the jump-pointer prefetching mechanism that is able to prefetch more than one node ahead. Here we compare the performance achieved by the push model with that of jump-pointer prefetching. The results are shown in Figure 12. Roth et al. suggested three jump-pointer implementations: hardware, software, and hardware-software cooperative. We adopted the software-hardware cooperative approach. All jump-pointers are installed eight nodes ahead. Ideally, the jump-pointer scheme can tolerate any amount of latency as long as the jump-pointer interval is set appropriately.[4] In contrast, as the analysis shown in Section 2.2, the performance of the push mechanism depends on three parameters: the amount of computation in each iteration, the DRAM access time, and round-trip memory latency. Therefore, for treeadd, health and perimeter, the jump-pointer prefetching scheme achieves higher speedup than the push mechanism. Since bh has long computation between node accesses, both schemes eliminate the memory stall time completely. For em3d, the push architecture also perform comparably to jump-pointer prefetching. Jump-pointer prefetching does not bring more performance advantage than sequential pull-based prefetching discussed above because em3d has some natural parallelism. There are two major deficiencies in jump-pointer prefetching.

[3]We tested mcf on the 1_PFE architecture where all prefetches are issued from the main memory level, and observed close to 10% speedup.

[4]Determining a suitable interval for each application is not a trivial task. They do not provide a mechanism to adapt the jump-pointer interval on an application basis in their work.

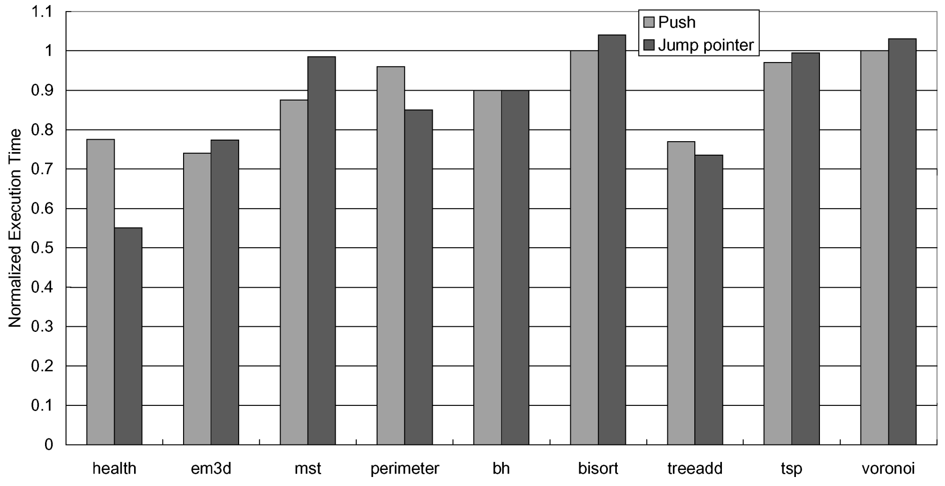


Fig. 12. Performance comparison between the push model and the jump-pointer prefetching mechanism.

First, for data sets that are traversed only once, such as mst, jump-pointers need to be installed during the LDS creation phase. If the LDS creation order is different from the traversal order, jump-pointer prefetching cannot generate effective prefetches. This is why the push model outperforms the jump-pointer scheme for mst. The second factor that limits the effectiveness of jump-pointer prefetching is the nontrivial overheads incurred by jump-pointer creation. If the performance gain from jump-pointer prefetching cannot compensate this overhead, jump-pointer prefetching could actually degrade performance. That is the reason for the adverse effect caused by jump-pointer prefetching for tsp, bisort, and voronoi. Both bisort and tsp have dynamically changing structures, and voronoi has more array/scalar loads than pointer loads. Even though the push architecture does not show performance benefit for these applications either, it does not incur significant run-time overheads as jump-pointer prefetching. Overall, the push architecture performs comparably to jump-pointer prefetching. Jump-pointer prefeching only shows 1.6% average performance advantage over the push model.

Since bisort, mcf, and voronoi do not benefit from the push model, we omit these applications in the following discussion.

4.3.2 *Effect of the PFE Data Cache and Throttle Mechanism.* Recall that the push architecture uses two mechanisms to avoid redundant and early prefetches: the PFE data cache and throttle mechanism. The result presented above show the combined effect of both features. In this section, we evaluate the effect of the PFE data cache and throttle mechanism separately in Figure 13. Push_base is a plain push model with no data caches or throttle mechanism. Push_throttle is push_base with throttling and push_buffer is push_base with data caches in the L2 and memory PFEs. The performance impact from these two techniques are not exclusive. The data caches can speed up PFE execution, while throttling slows down the PFE execution when they run too far

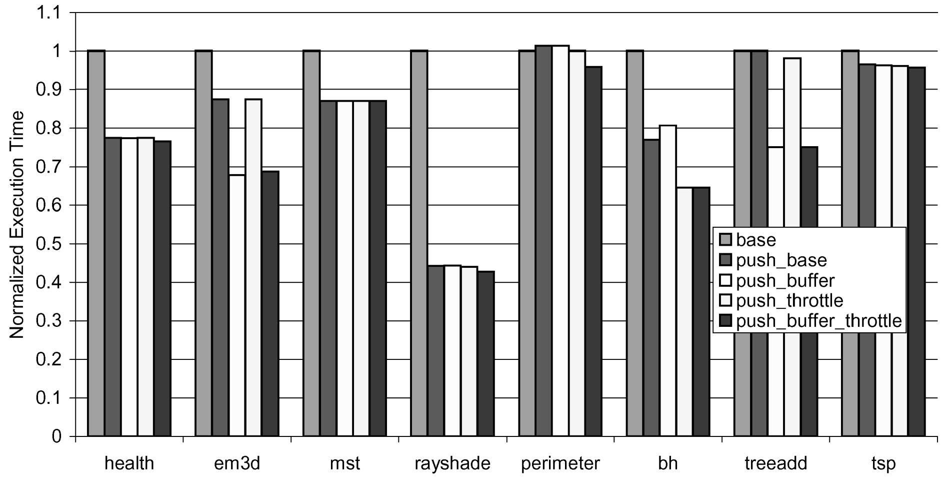


Fig. 13. Effect of the PFE data cache and throttle mechanism.

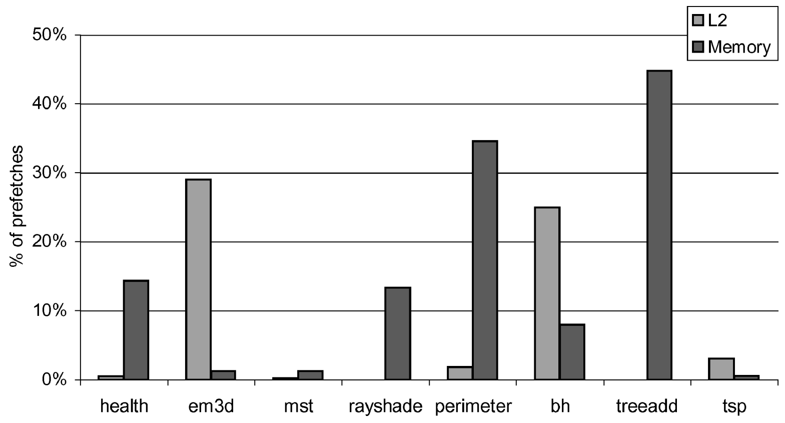


Fig. 14. Redundant prefetches from the L2/memory levels for push_base. The y-axis shows the percentage of redundant prefetches, which is obtained by dividing the number of redundant prefetches by the total number of prefetches.

ahead. Therefore, push_buffer_throttle shows the combined effect of both features, which is the previously presented results in Figure 11.

From Figure 13 we can see that em3d and treeadd benefit most from data caches. Push_base is only able to reduce execution time by 12% for em3d. Adding a data cache (push_buffer) further reduces execution time by 20%. Treeadd does not show performance improvement for push_base, but push_buffer reduces execution time by 25%. Perimeter does not see performance improvement comparing push_base and push_buffer. However, adding this feature on top of the throttle mechanism (i.e., push_buffer_throttle and push_throttle) does give performance improvement.

Em3d, perimeter, and treeadd have 30%, 33%, and 45% of prefetches that are redundant as shown in Figure 14. Bh also has a significant amount of redundant

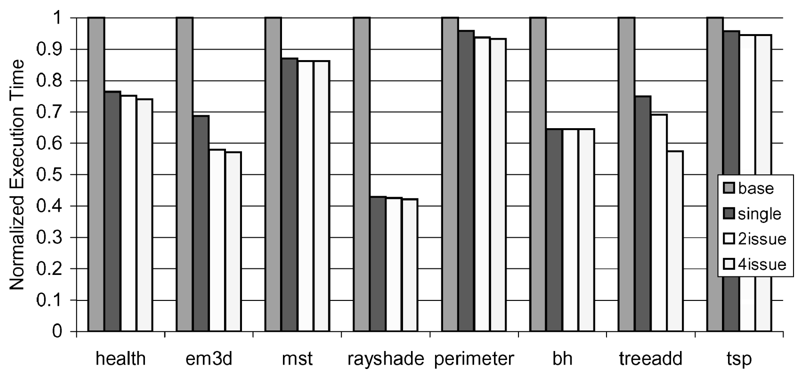


Fig. 15. Effect of wider issue PFEs.

prefetches (33%). The L2/memory PFE data caches are able to capture between 70% and 100% of these redundant prefetches. Treeadd, bh, and perimeter are tree traversals in depth-first order, which can cause redundant prefetches as mentioned in Section 3. Em3d simulates electromagnetic fields and processes lists of interacting nodes in a bipartite graph. The intersection of these lists creates locality, which results in redundant prefetches. Note that redundant prefetches do not affect the performance of bh because the PFEs already issue prefetch requests far ahead of the CPU for the push_base configuration (93% of prefetched blocks are replaced before accessed).

Figure 13 shows that the throttle mechanism has the most impact on bh (push_base vs. push_throttle). Bh has long computation between node accesses. So the PFE runs too far ahead of the CPU for push_base. 93% of prefetched cache blocks are replaced before accessed by the CPU. The proposed throttle mechanism successfully prevents early prefetches. Nearly 100% of prefetched cache blocks are accessed by the CPU for push_throttle. Push_base reduces execution time by 23%, and push_throttle further reduces it by 13%. It is surprising that push_base is able to reduce execution time by 23% even though most prefetched cache blocks are replaced from the prefetch buffer. Push_base obtains speedup because of better L2 cache performance compared to the base configuration (92% of L2 misses are eliminated). Since the push model deposits data in both the L2 cache and the prefetch buffer, blocks replaced from the prefetch buffer can still be resident in the L2 cache at the time the CPU accesses them.

4.3.3 *Effect of Wider Issue PFEs.* The results thus far show that a single-issue PFE processor is sufficient to produce significant performance improvements for the push model. Figure 15 shows the execution time of a single-issue, 2-issue and 4-issue PFE normalized to the base case without prefetching. From these results, we see that increasing the issue width produces noticeable improvements for several benchmarks, particularly em3d and treeadd. Em3d has a large traversal kernel that includes a nested loop with eight loads. Some of the loop control instructions and loads can be issued independently. Increasing the issue width to two further reduces the execution time by 13%. Treeadd

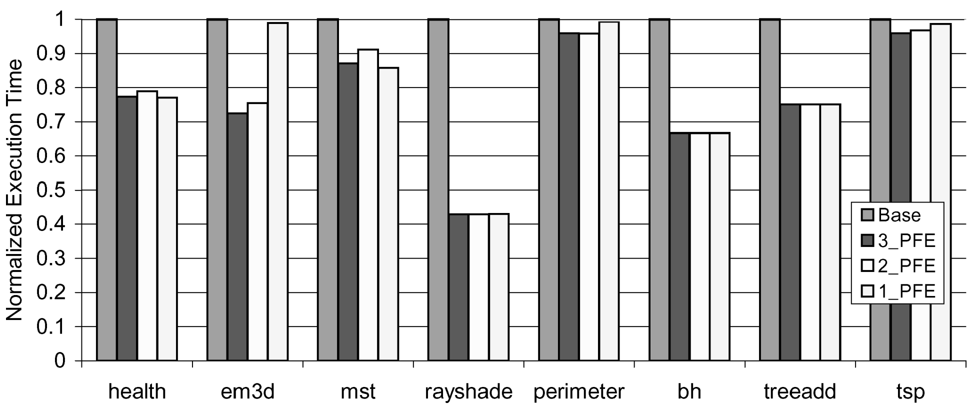


Fig. 16. Variations of the push architecture.

traverses a binary tree in depth-first order and loads a value associated with each node. Instructions that manipulate the stack can be issued independently. A 2-issue processor reduces the execution time by 10% more compared to a single-issue one. The execution time is further reduced by 5% after increasing the issue width from two to four. Note that we do not manually schedule instructions to take advantage of the wide issue capability, thus further performance improvement might be achieved if instruction scheduling is considered. Overall the modest complexity of increasing the issue width for statically scheduled PFE processors seems worth it.

4.3.4 *Number of PFEs.* This section evaluates the performance of various push architectures discussed in Section 3.5: 3_PFE, 2_PFE, and 1_PFE. 3_PFE attaches the PFE to each level of the memory hierarchy, which is the push architecture examined so far. 2_PFE attaches the PFE to the L1 and main memory levels, which performs pull-based prefetching until a recurrent load misses in the L2 cache. 1_PFE only uses one engine at the main memory level, so all prefetches are issued from the bottom of the memory hierarchy. Figure 16 shows execution time of these three architectures normalized to the base configuration.

From Figure 16 we see that the performance of 2_PFE is comparable to 3_PFE. This indicates that 3_PFE gains most of its performance from pushing data up to the L2 level. 1_PFE achieves similar performance to 3_PFE and 2_PFE for all benchmarks except for em3d. As mentioned in Section 3.5, 3_PFE issues most of the prefetches at the L2 level for em3d because 80% of L1 misses are satisfied by the L2 cache. For the 1_PFE architecture, all prefetches are now issued at the main memory level instead. If a LDS node exists in the L2 cache, it takes $r1 + a1 + x1$ cycles ($r1$: sending a request from L1 to L2; $a1$: L2 access; $x1$: transferring a cache block back to L1) to fetch a node up to the processor. For 1_PFE, the latency between recurrent prefetches is the DRAM access time, which is larger than $r1 + a1 + x1$ for the configuration tested (48 vs. 18 cycles). In this case, 1_PFE is not able to produce any prefetching effect. 2_PFE can deliver similar performance to 3_PFE because it simply performs pull-based prefetching. For em3d, the pull model performs comparably to the push model as shown in Figure 11.

From this experiment we know that 2_PFE achieves comparable performance to 3_PFE for all benchmarks. 1_PFE only needs one prefetch engine but it performs poorly if the L2 cache is able to capture most of the L1 misses, like em3d. As mentioned in Section 3.5, 2_PFE only needs one PFE if the CPU is a multithreading processor, which is the trend for future processors [Gwennap 1998]. Therefore, 2_PFE is the best design choice considering both cost and performance.

5. RELATED WORK

Early data prefetching research focused on array-based applications with regular access patterns. Hardware prefetching detects array access strides from the address history at run time [Baer and Chen 1991; Fu and Patel 1992; Jouppi 1990]. Software prefetching [Callahan et al. 1991; Klaiber and Levy 1991; Mowry et al. 1992; Porterfield 1989] exploits compile-time information to insert prefetch instructions in a program. Correlation-based prefetching [Alexander and Kedem 1996; Joseph and Grunwald 1997] also relies the address history to predict future references, but they can capture complex access patterns. The prediction accuracy relies on the size of the prediction table and stable access patterns.

The Spaid scheme proposed by Lipasti et al. [1995] is a compiler-based pointer prefetching mechanism. It inserts prefetch instructions for the objects pointed by pointer arguments at call sites. Luk and Mowry [1996] propose three compiler-based prefetching algorithms, greedy, history-pointer, and data-linearization prefetching. Chilimbi et al. [1999a, 1999b] reorganize data layouts to improve cache performance for irregular applications. Zhang and Torrellas [1995] use object information to guide prefetching for irregular applications in shared-memory multiprocessors. Mehrotra and Harrison [1996] extend stride detection schemes to capture both linear and recurrent access patterns. Roth et al. [1998] and Roth and Sohi [1999] propose a dynamic scheme to capture LDS traversal kernels and a jump-pointer prefetching framework to overcome the pointer-chasing problem. Ideally, the jump-pointer scheme can tolerate any amount of latency as long as the jump-pointer interval is set appropriately. However, determining a suitable interval for each application is not a trivial task. Roth et al. do not provide a mechanism to adapt the jump-pointer interval on an application basis. Moreover, there are several limitations on jump-pointer prefetching. First, the jump-pointer scheme incurs nontrivial run-time overhead, therefore, it could cause adverse effects on performance for applications with traversal patterns that is not suitable for jump-pointer prefetching (e.g., dynamically changing data structures). Even though the push architecture does not work well for highly dynamic structures either, we do not degrade performance. Second, the performance of jump-pointer is affected by the number of traversals performed on data sets since it requires one pass to install jump pointers. For applications that traverse data sets only once, jump-pointers have to be installed during structure construction. If the construction order is different from the traversal order, jump-point prefetching is not able to generate correct future addresses. The push architecture does not have this limitation. Third, since jump-pointer prefetching relies on earlier traversals to install jump pointers, it does not work well if traversal orders change frequently (e.g., tree searching). In contrast, the push architecture executes traversal kernels to generate future addresses, therefore, it can still provide correct prefetches even traversal orders are not fixed. Karlsson et al. [1999] presents a prefetch array approach, which aggressively prefetches all possible nodes a few iterations ahead. The downside of this approach is that it could issue many unnecessary prefetches.

Several studies [Kang et al. 1999; Oskin et al. 1998; Patterson et al. 1997] also combine processing power and memory in the same chip. The main function of DRAM processors is performing computation. Impulse [Carter et al. 1999] provides configurable physical address remapping in the memory controller to improve bus and cache utilization. The memory controller is also capable of prefetching data. But they only prefetch next cache line, and data are not pushed up the memory hierarchy as proposed in this work. Concurrent with this study, Hughes [2002] evaluates memory-side prefetching in multiprocessor systems. His scheme does not provide solutions for two important design issues of the push model: the interaction protocol among prefetch engines at different memory modules and a mechanism to synchronize the CPU and PFE execution. Solihin et al. [2002] also propose a memory-side prefetching scheme. They adopt the push model proposed in this work. They employ correlation prefetching and target at general irregular applications instead of linked data structures as this work. Even though they solve many problems in conventional correlation prefetching, their performance is still limited if repeated access patterns are absent. In contrast, our scheme does not rely on the past address history for future address prediction.

Some work also proposes using a separate processor for memory access. Structured memory access [Pleszkun and Davidson 1983] and the decoupled access execute [Smith 1982b] try to overlap demand memory requests with computation. VanderWiel and Lilja [1999] proposes a separate processor for prefetching purposes for regular applications. Recently, several studies [Annavaram et al. 2001; Collins et al. 2001; Dundas and Mudge 1997; Luk 2001; Roth and Sohi 2001; Sundaramoorthy et al. 2000; Zilles and Sohi 2001] suggest using pre-execution to improve cache performance for irregular applications. The idea of pre-execution is to execute a sequence of instructions (speculative slice) early and speculatively to hide memory latency. They either employ a separate processor at the L1 level to execute a speculative slice or simply invoke a helper thread if the CPU is a multithreading processor. This is essentially the pull model that we evaluate, and it can be limited by the conventional pull_based data movement.

6. CONCLUSIONS

In this paper, we propose a cooperative hardware/software prefetching framework for linked data structures—*the push architecture*. The push architecture uses a prefetch engine (PFE) to execute LDS traversal kernels (instructions that traverse LDSs) ahead of the CPU, thus successfully generating future addresses. LDS traversal kernels are constructed statically. The PFE is attached to each level of the memory hierarchy. Cache blocks accessed by these processors are pushed up to the CPU. The push model decouples the pointer dereference (obtaining the next node address) from the transfer of the current node up to the processor and allows implementations to pipeline these two operations. In this way, the latency between recurrent prefetches is reduced to only the DRAM access time as opposed to round-trip memory latency (including latency to access L1/L2 caches and bus transfer) in a pull-based prefetching scheme.

Our simulation results show that the push architecture is able to reduce 13% to 23% of the overall execution time for applications with very tight loops, which the traditional pull model is not able to run ahead of CPU to give significant performance improvement. Tree traversals benefit most from adding a small data cache in the L2/memory PFEs (e.g., treeadd). We have also shown that the proposed throttle mechanism successfully adjusts the prefetch distance to avoid early prefetches. For applications with enough computation between node accesses, the push architecture is able to achieve performance comparable to a perfect memory system. Simulations also show that the 2_PFE architecture, which only attaches the PFE to the L1 and main memory levels, performs comparably to 3_PFE, which attaches the PFE to each level of the memory hierarchy. We also compare the push architecture against jump-pointer prefetching, which is the state-of-art pulled-based prefetching mechanism, and find that the push architecture provides comparable average performance with jump-pointer prefetching.

REFERENCES

Alexander, T. and Kedem, G. 1996. Distributed predictive cache design for high performance memory system. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*. 254–263.

Annavaram, M. M., Patel, J. M., and Davidson, E. S. 2001. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 52–61.

Baer, J.-L. and Chen, T.-F. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 Conference on Supercomputing*. 176–186.

Burger, D. C., Austin, T. M., and Bennett, S. 1996. *Evaluating Future Microprocessors the Simplescalar Tool Set.* Tech. Rep. 1308, Computer Sciences Department, University of Wisconsin–Madison. July.

Callahan, D., Kennedy, K., and Porterfield, A. 1991. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. 40–52.

Carter, J., Hsieh, W., Stoller, L., Swanson, M., and Zhang, L. 1999. Impulse: Building a smarter memory controller. In *Proceedings of 5th Symposium High-Performance Computer Architecture*. 70–79.

Chilimbi, T., Larus, J., and Hill, M. 1998. *Improving Pointer-Based Codes through Cache-Concious Data Placement.* Tech. Rep. CSL-TR-98-1365, University of Wisconsin, Madison. March.

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999a. Cache-conscious structure definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*. 13–24.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999b. Cache-conscious structure layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*. 1–12.

COLLINS, J. D., WANG, H., TULLSEN, D. M., CHRISTOPHER, H. J., LEE, Y. F., LAVERY, D., AND SHEN, J. P. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 14–25.

DUNDAS, J. AND MUDGE, T. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the ACM International Conference on Supercomputing*. 176–186.

FU, J. W. C. AND PATEL, J. H. 1992. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. 102–110.

GWENNAP, L. 1998. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*.

HUGHES, C. J. 2002. Prefetching linked data structures in systems with merged dram-logic. M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 252–263.

JOUPPI, N. P. 1990. 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*. 364–373.

KANG, Y., HUANG, W., YOO, S.-M., KEEN, D., GE, Z., LAM, V., PATTNAIK, P., AND TORRELLAS, J. 1999. Flexram: Toward an advanced intelligent memory system. In *Proceedings of the 1999 International Conference on Computer Design*. 192–201.

KARLSSON, M., DAHLGREN, F., AND STENSTROM, P. 1999. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of 6th Symposium High-Performance Computer Architecture*. 206–217.

KESSLER, R. E. 1999. The alpha 21264 microprocessor. *IEEE Micro*. 34–36.

KLAIBER, A. C. AND LEVY, H. M. 1991. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 43–53.

KOLB, C. The rayshade user's guide. In http://graphics.stanford.edu/- cek/-rayshade.

KROFT, D. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*. 81–87.

LEBECK, A. AND WOOD, D. 1994. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*. 15–26.

LEIBSON, S. 2000. Xscale (strongarm-2) muscles. *Microprocessor Report*.

LIPASTI, M. H., SCHMIDT, W. J., KUNKEL, S. R., AND ROEDIGER, R. R. 1995. Spaid: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 252–263.

LUK, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 40–51.

LUK, C.-K. AND MOWRY, T. C. 1996. Compiler based prefetching for recursive data structure. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. 222–233.

MEHROTRA, S. AND HARRISON, L. 1996. Examination of a memory access classification scheme for pointer-intensive and numeric program. In *Proceedings of the 10th International Conference on Supercomputing*. 133–139.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System*. 62–73.

OSKIN, M., CHONG, F. T., AND SHERWOOD, T. 1998. Active pages: a computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 192–203.

PATTERSON, D., ANDRESON, T., CARDWELL, N., FROMM, R., KEATON, K., KAZYRAKIS, C., THOMAS, R., AND YELLICK, K. 1997. A case for intelligent ram. *IEEE Micro*. 34–44.

PLESZKUN, A. R. AND DAVIDSON, E. S. 1983. Structured memory access architecture. In *Proceedings of International Conference on Parallel Processing*. 461–471.

PORTERFIELD, A. K. 1989. Software methods for improvement of cache performance on supercomputer applications. Ph.D. Thesis, Department of Computer Science, Rice University.

ROGER, A., CARLISLE, M., REPPY, J., AND HENDREN, L. 1995. Supporting dynamic data structures on distributed memory machines. *ACM Trans. Program. Lang. Syst. 17,* 2 (March).

ROTH, A., MOSHOVOS, A., AND SOHI, G. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. 115–126.

ROTH, A. AND SOHI, G. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. 111–121.

ROTH, A. AND SOHI, G. 2001. Speculative data-driven multithreading. In *Proceedings of 7th Symposium High-Performance Computer Architecture*. 134–143.

SMITH, B. 1982a. Architecture and applications of the hep multiprocessor computer system. In *Proceedings of the International Society for Optical Engineers*. 241–248.

SMITH, J. E. 1982b. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*. 112–119.

SOHI, G. 1990. Instruction issue logic for high performance, interruptable, multiple functional unit, pipelined computers. *IEEE Trans. Comput. 39*, 3 (March), 349–359.

SOLIHIN, Y., LEE, J., AND TORRELLAS, J. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 171–182.

SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. 2000. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. 257–268.

VANDERWIEL, S. AND LILJA, D. J. 1999. *A Compiler-Assisted Data Prefetch Controller.* Tech. Rep. ARCTiC-99-05, Department of Electrical and Computer Engineering, University of Minnesota. May.

YANG, C. AND LEBECK, A. R. 2000. Push vs. pull: Data movement for linked data structures. In *Proceedings of the ACM International Conference on Supercomputing*. 176–186.

ZHANG, Z. AND TORRELLAS, J. 1995. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 188–200.

ZILLES, C. B. AND SOHI, G. 2001. Execution-base prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 2–13.

Received July 2003; revised May 2004 and October 2004; accepted October 2004