A Programmable Memory Hierarchy for Prefetching Linked Data Structures

Chia-Lin Yang[1] and Alvin Lebeck[2]

[1] National Taiwan University, 1 Roosevelt Rd. Sec. 4, Taipei, Taiwan
yangc@csie.ntu.edu.tw
[2] Duke University, Durham, NC 27708, USA
alvy@cs.duke.edu

Abstract. *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 use a programmable processor, a prefetch engine (PFE), at each level of the memory hierarchy to cooperatively execute instructions that traverse a linked data structure. Cache blocks accessed by the processors at the L2 and memory levels are proactively pushed up to the CPU.*
We look at several design issues to support this programmable memory hierarchy. We establish a general interaction scheme among three PFEs and design a mechanism to synchronize the PFE execution with the CPU. Our simulation results show that the proposed prefetching scheme can reduce up to 100% of memory stall time on a suite of pointer-intensive applications, reducing overall execution time by an average 19%.

1 Introduction

The widening gap between processor cycle time and main memory access time makes techniques to alleviate this disparity essential for building high performance computer systems. Caches are recognized as a cost-effective method to bridge this gap. However, the effectiveness of caches is limited for programs with poor locality. Programs with regular access patterns can often employ prefetching techniques to hide memory latency. Unfortunately, these techniques are difficult to apply to pointer-intensive data structures because the address stream lacks regularity and there are data dependencies between elements in the structure.

The lack of address regularity makes it difficult for conventional prefetching techniques to predict future addresses. The data dependence is a result of pointer indirection. Pointer dereferences are required to generate addresses for successive elements in a linked data structure (LDS). This is commonly called the pointer-chasing problem. This serialization hinders efforts to choose appropriate prefetch distances [1] so that memory latency can be fully overlapped with computation.

We recently proposed a novel data movement model—called push—to overcome the above limitations [2]. The push model performs pointer dereferences at lower levels of the memory hierarchy and pushes data up to the processor.

This decouples the pointer dereference from the transfer of the current LDS element up to the processor. Implementations can pipeline these two operations and eliminate the request-response delay required for a conventional pull-based technique where the processor fetches an LDS element before requesting the next element. To realize this push model, we attach a prefetch engine (PFE) to each level of the memory hierarchy. The prefetch engines execute instructions that access LDS elements (LDS traversal kernels), and cache blocks accessed by the prefetch engines are pushed up to the CPU. An important aspect of this model is that it is not required for correct program execution, it is simply a performance enhancement similar to other non-binding prefetching techniques.

The contribution of this paper is that we provide a general architectural solution for this novel data movement model. Our previous work [2] performs a preliminary performance evaluation using a limited implementation of the push model. The initial design can only support linked-list traversals, which simplify the interaction among prefetch engines. This paper presents a flexible implementation of the push model – the push architecture. First, we use a fully programmable processor in each prefetch engine to support a multitude of LDS traversal kernels instead of the specialized hardware used in the initial design. In this way, the push model can also be easily extended to prefetch other data structures, e.g., sparse matrix, or to execute speculative slices [3–5]. Second, to accommodate more sophisticated structures (e.g., trees) we establish a general interaction scheme among three PFEs, particularly how a PFE suspends and resumes execution. Third, we present a throttle mechanism to synchronize the CPU and PFE execution. Fourth, we evaluate two variations of the push architecture which require less hardware resources. One attaches the PFE to the L1 and main memory levels; the other uses only one PFE at the main memory level.

Our simulations show that a simple, single-issue, in-order processor (e.g., ARM-like core) is sufficient to provide significant speedups for most of our benchmarks across a variety of linked data structures. The proposed prefetching scheme reduces the execution time between 13% to 25% for applications that the traditional pull method can not achieve significant speedup. In the specific case of list-based applications, the programmable PFE is able to deliver performance within 10% of the specialized hardware. We also find that the push architecture using the PFE only at the L1 and main memory levels can achieve performance close to using the PFE at each level of the memory hierarchy.

The rest of this paper is organized as follows. Section 2 provides background on the push model. Related work is discussed in Section 3. Section 4 describes the design details of our programmable memory hierarchy. Section 5 presents our experimental methods and results are presented in Section 6. We conclude in Section 7.

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

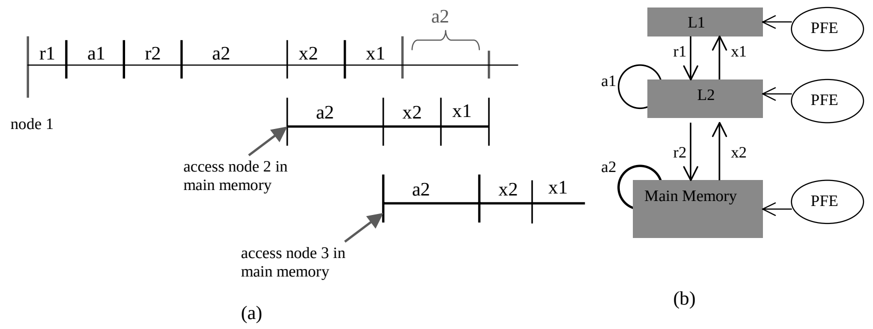

Fig. 1. The Push Model

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. The latency between recurrent prefetches in a pull-based prefetching scheme is the round-trip memory latency.

In contrast, the push model performs pointer dereferences at the lower levels of the memory hierarchy by executing traversal kernels in micro-controllers 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 as shown in Figure 1(a). In this way, we are able to request subsequent nodes in the LDS much earlier than a traditional system. In the push model, node i+1 arrives at the CPU a2 cycles (the DRAM access time) after node i. From the CPU standpoint, the latency between node accesses is reduced from the round-trip memory latency to DRAM access time.

To realize the push model we attach a prefetch engine to each level of the memory hierarchy model as shown in Figure 1(b). The prefetch engines (PFE) execute traversal kernels independent of CPU execution. Cache blocks accessed by the prefetch engine in the L2 or main memory level are pushed up to the CPU and stored in a prefetch buffer. The prefetch buffer is a small, fully-associative cache, which can be accessed in parallel with the L1 cache. Our previous work realizes this push model only for list-based applications [2]. Although the existing design could support a few different linked list traversals, it is not flexible enough to accommodate more sophisticated structures (e.g., trees). In this paper, we present a flexible implementation of the push model which is able to support a multitude of LDS traversal kernels.

3 Related Work

Early data prefetching research focuses on array-based applications with regular access patterns [6–9]. Correlation-based prefetching [10, 11] can capture complex access patterns from the address history, but the prediction accuracy relies on the size of the prediction table and stable access patterns.

Several studies [12–14] explore data structure information to insert prefetch instructions at compile time for irregular applications. Chilimbi et al. [15, 16] seek to improve cache performance of pointer-based applications by reorganizing data layouts. Mehrotra et al. [17] extend stride detection schemes to capture both linear and recurrent access patterns. Roth et al. [18, 19] propose a dynamic scheme to capture LDS traversal kernels and a jump-pointer prefetching framework to overcome the pointer-chasing problem. The performance of the jump-pointer approach is limited for applications which change the LDS structure or traversal order frequently. Applications that have short LDS or only traverse the LDS for few times have also limited benefit from jump-pointer prefetching. Karlsson et al. [20] present 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 [21–23] also combine processing power and memory in the same chip. The push architecture differs from these studies in that the PFEs do not perform computation except for address calculation and value comparison for control flow. The memory controller in Impulse [24] 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 paper. Concurrent with this study, Hughes [25] 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.

Recently, several studies [26, 3, 4, 27, 5, 28–30] suggest using pre-execution to improve cache performance for irregular applications. They either employ a seperate 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. Pre-execution techniques could be used with the push model by executing the speculative slices on the PFEs in the memory hierarchy. Crago et al. [31] propose a hierarchical decoupled architecture that adds a processor between each level of the memory hierarchy. Their framework is still based on the traditional pull model, and they only study array-based applications.

4 The Push Architecture

In this section, we present an architectural design to realize the push model. We adopt a 5 stage pipeline, in-order issue general processor core as the PFE. The PFE implements a root register to store the root address of the LDS being traversed. A write to the root register activates the PFE. The L2/memory PFE contains a TLB for address translation and 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. The details of the

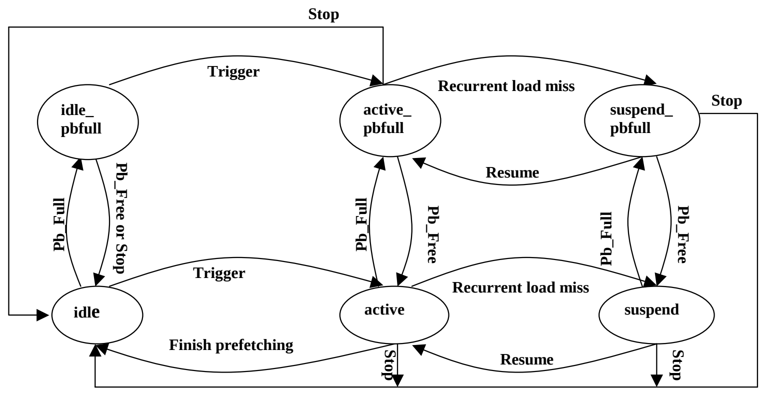

Fig. 2. Interaction Profocol among the PFEs and CPU

PFE microarchitecture can be found in [32]. Below we describe the interaction scheme among 3 level PFEs and a mechanism to synchronize the CPU and PFE execution. We also discuss two variations of the push architecture.

4.1 Interaction Among Prefetch Engines

The main design challenge of the push architecture lies in the interaction among the CPU and three PFEs. In this section, we focus on how three PFEs work together to perform prefetching. The next section addresses how the PFEs synchronize with the CPU execution. The complete interaction protocol is summarized in Figure 2. Below we elaborate on the operations of this protocol.

The L1 PFE is triggered when the root address of the LDS being traversed is written into the root register (i.e. from the idle to active state). The L1 PFE continues execution until a recurrent load (a load that generates the address of the next LDS element) misses in the L1 cache. 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. The PFE uses a new flavor of load instruction, ld_recurrent, to distinguish a recurrent load from other loads. When a recurrent load causes a L1 miss, the cache controller signals the L1 PFE to suspend execution (i.e. from the active to suspend state). 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.

Consider traversing a binary tree in depth-first order, as shown in Figure 3. The assembly code is the traversal kernel. Assume that the recurrent load x (in Figure 3(a)), between node 1 and 2 in Figure 3(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 nodes

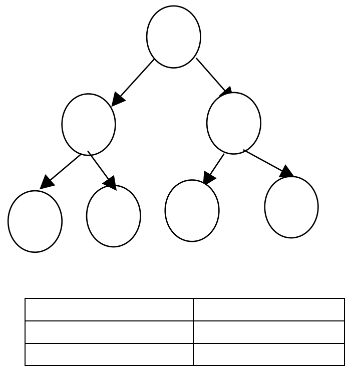

Fig. 3. Tree Traversal

2, 3 and 4. The L1 engine should resume execution to prefetch the remaining tree nodes after the memory PFE finishes prefetching.

We assume a dedicated bus, called the PFE Channel, for communication between the PFEs. When the PFE finishes prefetching, it sends a Resume token up the memory hierarchy through the PFE Channel. The first PFE that is in the suspend state keeps the token and resumes execution (i.e. from the suspend to active state). In this example, the L1 PFE resumes execution after receiving Resume token sent by the memory PFE. To correctly prefetch the remaining tree nodes, the L1 PFE needs to set its current program counter (PC) to 0x400998 (i.e. the recurrent load to node 5) and ensure the register values are valid. Because 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.

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 suspend execution, the PFE uses these two pieces of information to find the resume PC and restore the register values. First, the RPC is used to access the Resume PC Table (see Figure 3(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 implementation details are described in [32].

4.2 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 a 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 [33]. 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 so synchronization between the CPU and PFEs needs to occur.

When the prefetch buffer is full, the PFEs could be running too far ahead of the CPU or performing useless prefetching (e.g., mis-predicted or late prefetches), so the active PFE should suspend execution at this point. The L1 cache controller broadcasts a Pb_Full message to the PFEs through the PFE Channel once it detects that prefetch buffer is full. The active engine suspends execution after observing the Pb_Full message (i.e., from the active to active_pbfull state).

If the PFEs are running ahead of the main execution, the CPU will eventually access the prefetch buffer. Once an entry is freed up, the L1 cache controller broadcasts a Pb_Free message, and the active engine resumes execution (i.e., from the active_pbfull to active state). 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. If the prefetch buffer is full because of useless prefetches, most of the time, the CPU will never access data in the prefetch buffer, and the PFE should start at a new point. We use a recurrent load miss from the CPU execution as a synchronization point. The implementation details can be found in [32].

4.3 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.

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. For LDS elements that exist in the L2 cache, the push model brings data to the L1 level only $r1 + x1$ cycles earlier than the pull model ($r1$: time to send a request from L1 to L2; $x1$: time to transfer a cache block from L2 to L1). 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 [4, 5]. In this way, 2_PFE only needs to employ one PFE at the main memory level.

In the 1_PFE architecture, the root address of the LDS being traversed is written into the root register of the memory PFE. 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 (the DRAM access time vs. round-trip L2 latency). So 1_PFE may not be able to achieve any prefetching effect because the memory PFE is very likely to run behind the CPU.

5 Methodology

To evaluate the push model we modified the SimpleScalar simulator [34] to include our PFE implementation. Our base model uses a 4-way superscalar, out-of-order processor. The memory system consists of a 128-entry TLB, a 32KB, 32-byte cache line, 2-way set-associative first level data and instruction caches and a 512KB, 64-byte cache line, 4-way set associative unified second level cache. We implement a hardware managed TLB with a 30 cycle miss penalty. Both the L1 and L2 caches are lock-up free caches that can have up to eight outstanding misses. The L1 cache has 2 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 [35], which has a 800MHz CPU and 60ns DRAM access time (i.e., 48 cycles). We also vary the memory system parameters to evaluate the push model for future processors. Simulation results can be found in [32].

To evaluate the performance of the push model, we use an in-order, single-issue processor as the PFE. 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. 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 [36] to identify kernels that contribute the most cache misses, and construct only these kernels by hand.

We evaluate the push model using the Olden pointer-intensive benchmark suite [37] and Rayshade [38]. 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 [18, 19, 13]. We omit the power application because it has a low (1%) miss rate. Rayshade is a real-world graphics application that implements a raytracing algorithm. We evaluate the push model performance for both a perfect and 128-entry TLB. For this set of benchmark tested, a 128-entry TLB has less than 1% impact on the push model performance except for health. Therefore, we only present simulation results assuming a perfect TLB in this paper.

6 Simualtion Results

In this section, we present simulation results that demonstrate the effectiveness of the proposed prefetching scheme. We first evaluate the performance improvement of the push model on the 3_PFE architecture. We then examine the effect of the PFE data cache and throttle mechanism. We also compare the performance of the push model using a programmable PFE to that of a specialized PFE for list-based applications. We finish by evaluating the performance of the 2_PFE and 1_PFE architecture.

Performance Comparison Between The Push and Pull Model
Figure 4 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 2 components, memory stall time and computation time. We obtain the computation time by assuming 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 41% 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 (rayshade and em3d), the pull model is able to achieve significant speedup (25% and 39% execution time reduction), but the push model still outperforms the pull model (31% and 57% execution time reduction). For rayshade, the push model is able to reduce 89% of memory stall time, but the pull model can only reduce 62%. For bh, both the push and pull models reduce up to 100% of memory stall time.

Bisort and tsp dynamically change the data structure while traversing it. The prediction accuracy is low for this type of application because the PFEs will prefetch the wrong data after the CPU has modified the structure. For tsp, we are able to identify some traversal kernels that do not change the structure dynamically. The results presented here prefetch only these traversal kernels. 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

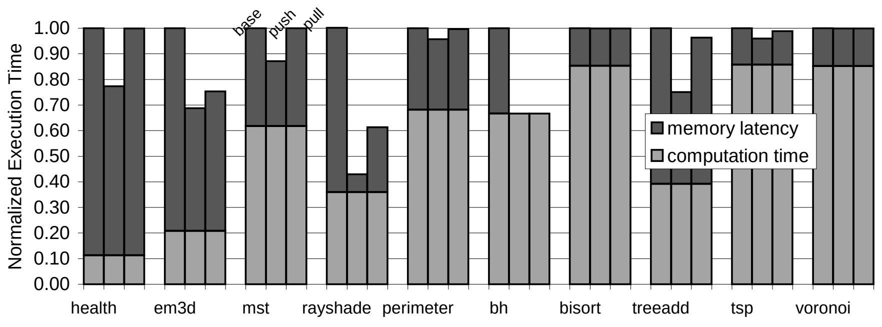

Fig. 4. Performance comparison between the push and pull model

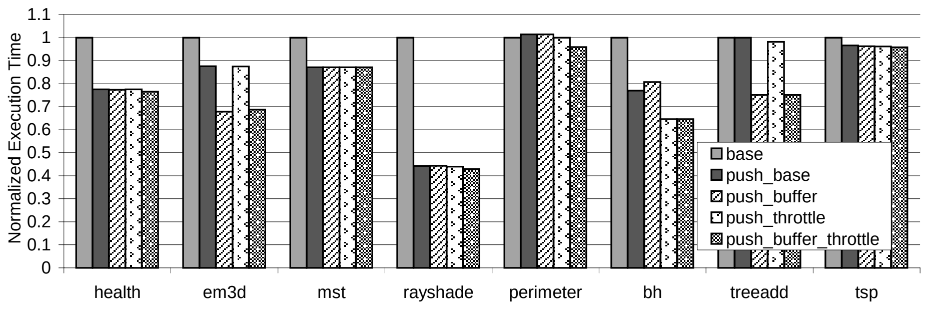

Fig. 5. Effect of the PFE Data Cache and Throttle Mechanism

the prediction accuracy is low (only 20% of prefetched cache blocks are useful). Voronoi uses pointers, but array and scalar loads cause most of the cache misses. Since we did not target these misses, we do not see any performance improvement for either the push or pull model. Since bisort and voronoi do not benefit from the push model, we omit these applications in the following discussion.

From these results we know that 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 (e.g., health and mst). For applications with enough computation between node accesses, the push model is able to achieve performance close to a perfect memory system, but the pull model is still not able to deliver comparable performance (e.g., rayshade).

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 results 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 5. 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 PEF execution when they run too far ahead. Therefore, push_buffer_throttle shows the combined effect of both features, which is the previously presented results in Figure 4.

From Figure 5 we 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. The L2/memory PFE data caches are able to capture between 80% to 100% of these redundant prefetches. Treeadd and perimeter are tree traversals in depth-first order, which generates redundant prefetches when the execution moves up the tree. Em3d simulates electro-magnetic 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 bh traverses an octree in depth-first order so it also has a significant amount of redundant prefetches (33%). However, bh does not benefit from the PFE data cache as shown in Figure 5 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 5 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.

Programmable vs. Specialized PFE

Our previous work proposes a specialized PFE for traversing linked lists. This specialized hardware traverses the entire linked list until a NULL pointer is encountered. The programmable PFE used in this paper has the flexibility advantage over the specialized hardware, but may not be able to deliver the same performance. Simulations show that for three list-based applications (health, em3d and rayshade), the programmable PFE is able to deliver performance within

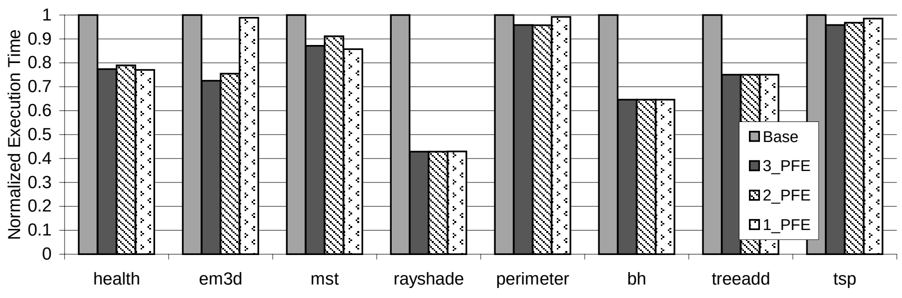

Fig. 6. Variations of the Push Architecture

10% of the specialized hardware. For rayshade, it even slightly outperforms a specialized PFE (1%). Rayshade always traverses the whole linked lists, but the fields of a LDS structure accessed cannot be determined statically. The specialized PFE is not able to handle such dynamic behavior as the programmable one.

Number of PFEs

This section evaluates the performance of various push architectures discussed in Section 4.3: 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 6 shows execution time of these three architectures normalized to the base configuration.

From Figure 6 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. For em3d, 80% of L1 misses are satisfied by the L2 cache. Therefore, most load instructions are resolved more slowly in the memory PFE than in the CPU (the DRAM access time vs. round-trip L2 latency). Thus 1_PFE is not able to produce any prefetching effect. Note that 1_PFE performs better than 2_PFE for mst. Mst implements hash table lookup so the list being traversed is very short. 1_PFE gets some performance advantage over 2_PFE by starting prefetching at the main memory level earlier.

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 4.3, 2_PFE only needs one PFE if the CPU is multithreaded [39]. Therefore, 2_PFE is the best design choice considering both cost and performance.

7 Conclusion

In this paper, we describe a flexible implementation of the push model, which overcomes the pointer-chasing problem by decoupling the pointer dereference from the transfer of the current node up to the processor. We use a programmable processor at each level of the memory hierarchy to cooperatively execute a LDS traversal kernel. The cache blocks accessed by these processors are pushed up to the CPU. We establish a general interaction scheme among three PFEs and propose a throttle mechanism to synchronize the CPU and PFE execution. This architecture could also be used to execute speculative slices or other compiler generated prefetch operations.

Our simulation results show that the push architecture is able to reduce 13% to 25% of the overall execution time for applications with very tight loops, while the traditional pull model is not able to run ahead of the CPU to give significant performance improvement. 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 performs comparably to 3_PFE.

We are currently investigating several extensions to this work. First, is to construct traversal kernels automatically, either by compiler or executable editor. We are also extending the push model to prefetch missing TLB entries and data structures other than LDS. Finally, we must examine issues surrounding context switches.

References

1. Klaiber, A.C., Levy, H.M.: An architecture for software-controlled data prefetching. In: Proceedings of the 18th Annual International Symposium on Computer Architecture. (1991) 43–53
2. Yang, C., Lebeck, A.R.: Push vs. pull: Data movement for linked data structures. In: Proceedings of the ACM International Conference on Supercomputing. (2000) 176–186
3. Collins, J.D., Wang, H., Tullsen, D.M., Christopher, H.J., Lee, Y.F., Lavery, D., Shen, J.P.: Speculative precomputation: Long-range prefetching of delinquent loads. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001) 14–25
4. Roth, A., Sohi, G.: Speculative data-driven multithreading. In: Proceedings of 7th Symposium High-Performance Computer Architecture. (2001) 134–143
5. Zilles, C.B., Sohi, G.: Execution-base prediction using speculative slices. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001) 2–13
6. Jouppi, N.P.: 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. (1990) 364–373

7. Baer, J.L., Chen, T.F.: An effective on-chip preloading scheme to reduce data access penalty. In: Proceedings of the 1991 Conference on SuperComputing. (1991) 176–186
8. Callahan, D., Kennedy, K., Porterfield, A.: Software prefetching. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV). (1991) 40–52
9. Mowry, T.C., Lam, M.S., Gupta, A.: Design and evaluation of a compiler algorithm for prefetching. In: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating System. (1992) 62–73
10. Joseph, D., Grunwald, D.: Prefetching using markov predictors. In: Proceedings of the 24th Annual International Symposium on Computer Architecture. (1997) 252–263
11. Alexander, T., Kedem, G.: Distributed predictive cache design for high performance memory system. In: Proceedings of the 2th International Symposium on High-Performance Computer Architecture. (1996)
12. Lipasti, M.H., Schmidt, W.J., Kunkel, S.R., Roediger, R.R.: Spaid: Software prefeteching in pointer- and call-intensive environments. In: Proceedings of the 28th Annual International Symposium on Microarchitecture. (1995)
13. Luk, C.K., Mowry, T.C.: Compiler based prefetching for recursive data structure. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII). (1996) 222–233
14. Zhang, Z., Torrellas, J.: Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture. (1995) 188–200
15. Chilimbi, T.M., Hill, M.D., Larus, J.R.: Cache-conscious struture layout. In: Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation. (1999) 1–12
16. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-conscious struture definition. In: Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation. (1999) 13–24
17. Mehrotra, S., Harrison, L.: Examination of a memory access classification scheme for pointer-intensive and numeric program. In: Proceedings of the 10th International Conference on Supercomputing. (1996) 133–139
18. Roth, A., Moshovos, A., Sohi, G.: Dependence based prefetching for linked data structures. In: Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII). (1998) 115–126
19. Roth, A., Sohi, G.: Effective jump-pointer prefetching for linked data structures. In: Proceedings of the 26th Annual International Symposium on Computer Architecture. (1999) 111–121
20. Karlsson, M., Dahlgren, F., Stenstrom, P.: A prefetching technique for irregular accesses to linked data structures. In: Proceedings of Sixth Symposium High-Performance Computer Architecture. (1999) 206–217
21. Patterson, D., Andreson, T., Cardwell, N., Fromm, R., Keaton, K., Kazyrakis, C., Thomas, R., Yellick, K.: A case for intelligent ram. IEEE Micro (1997) 34–44
22. Kang, Y., Huang, W., Yoo, S.M., Keen, D., Ge, Z., Lam, V., Pattnaik, P., Torrellas, J.: Flexram: Toward an advanced intelligent memory system. In: Proceedings of the 1999 International Conference on Computer Design. (1999) 192–201

23. Oskin, M., Chong, F.T., Sherwood, T.: Active pages: a computation model for intelligent memory. In: Proceedings of the 25th Annual International Symposium on Computer Architecture. (1998) 192–203
24. Carter, J., Hsieh, W., Stoller, L., Swanson, M., Zhang, L.: Impulse: Building a smarter memory controller. In: Proceedings of 5th Symposium High-Performance Computer Architecture. (1999) 70–79
25. Hughes, C.J.: Prefetching linked data structures in systems with merged dram-logic, master thesis. Technical Report UIUCDCS-R-2001-2221, Department of Computer Science, University of Illinois at Urbana-Champaign (2000)
26. Annavaram, M.M., Patel, J.M., Davidson, E.S.: Data prefetching by dependence graph precomputation. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001) 52–61
27. Sundaramoorthy, K., Purser, Z., Rotenberg, E.: Slipstream processors: Improving both performance and fault tolerance. (2000) 257–268
28. Luk, C.K.: Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001) 40–51
29. Collins, J., Tullsen, D., Wang, H., Shen, J.: Dynamic speculative precomputation. In: Proceedings of the 34st Annual International Symposium on Microarchitecture. (2001)
30. Moshovos, A., Pnevmatikatos, D., Baniasadi, A.: Slice processors: An implementation of operation-based prediction. In: Proceedings of the ACM International Conference on Supercomputing. (2001) 321–334
31. Crago, S.P., Despain, A., Gaudiot, J., Makhija, M., Ro, W., Sricastava, A.: A high-performance, hierarchical decoupled architecture. In: Proceedings of the Memory Access Decoupling for SuperScalar and Multiple Issue Architecture Workship. (2000)
32. Yang, C.L.: The Push Architecture: a Prefetching Framework for Linked-Data Structure. PhD thesis, Department of Computer Science, Duke University (2001)
33. Smith, B.: Architecture and applications of the hep multiprocessor computer system. In: Proceedings of the Int. Soc. for Opt. Engr. (1982) 241–248
34. Burger, D.C., Austin, T.M., Bennett, S.: Evaluating future microprocessors-the simplescalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison (1996)
35. Kessler, R.E.: The alpha 21264 microprocessor. IEEE Micro (1999) 34–36
36. Lebeck, A., Wood, D.: Cache profiling and the spec benchmarks: A case study. In: IEEE Computer. (1994) 15–26
37. Roger, A., Carlisle, M., Reppy, J., Hendren, L.: Supporting dynamic data structures on distributed memory machines. ACM Transactions on Programming Languages and Sytems **17** (1995)
38. Kolb, C.: The rayshade user's guide. (In: http://graphics.stanford.edu/- cek/-rayshade)
39. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture. (1995) 392–403