THE PUSH ARCHITECTURE: A PREFETCHING FRAMEWORK FOR LINKED DATA STRUCTURES

by

Chia-Lin Yang

Department of Computer Science
Duke University

Date: ______________________

Approved:

____
Dr. Alvin R. Lebeck, Supervisor

____
Dr. Jeffrey S. Chase

____
Dr. Gershon Kedem

____
Dr. Nikos P. Pitsianis

____
Dr. Xiaobai Sun

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2001

<u>ABSTRACT</u>

(Computer Science)

THE PUSH ARCHITECTURE: A PREFETCHING FRAMEWORK FOR LINKED DATA STRUCTURES

by

Chia-Lin Yang

Department of Computer Science
Duke University

Date: ____________________________
Approved:

____
Dr. Alvin R. Lebeck, Supervisor

____
Dr. Jeffrey S. Chase

____
Dr. Gershon Kedem

____
Dr. Nikos P. Pitsianis

____
Dr. Xiaobai Sun

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2001

Abstract

The widening performance gap between processors and memory makes techniques that alleviate this disparity essential for building high-performance computer systems. Caches are recognized as a cost-effective method to improve memory system performance. However, a cache's effectiveness can be limited if programs have poor locality. Thus techniques that hide memory latency are essential to bridging the CPU-memory gap.

Prefetching is a commonly used technique to overlap memory accesses with computation. Prefetching for array-based numeric applications with regular access patterns has been well studied in the past decade. However, prefetching for pointer-intensive applications remains a challenging problem. Prefetching linked data structures (LDS) is difficult because address sequences do not present the same arithmetic regularity as array-based applications and because data dependence of pointer dereferences can serialize the address generation process.

The push architecture proposed in this thesis is a cooperative hardware/software prefetching framework 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 suit of pointer-intensive applications, reducing overall execution time by an average 19%.

Acknowledgements

As I look back the past six years, my heart is filled with thankfulness. I thank God for those people who have helped me in this journey.

I would like to express my deepest appreciation to my advisor, Professor Alvin R. Lebeck. He guided me through every step of my Ph.D study with patience and encouragement. He has been a great mentor for me in these years. I hope I could have the same influence on my future students as he has on me.

I would like to thank Processor Jeffrey S. Chase, Professor Gershon Kedem, Dr. Nikos P. Pitsianis and Professor Xiaobai Sun for taking their time to serve in my Ph.D. committee and give me valuable comments on this thesis. I want to express my special thanks to Professor Xiaobai Sun for her friendship and encouragement throughout my years at Duke.

Special thanks to Dr. Barton Sano and Dr. Norman P. Jouppi for a great intern experience at Western Research Lab. I also like to thank Intel Foundation for supporting me to finish this thesis.

I would like to thank my friends, Srikanth Srinivasan, Mithuna Thottethodi, Wei Jin, Chong Xu, Rung-Huang Tsai, Yujuan Bao and Elizabeth Cherry. Their friendship has brought me many joys in these years. I also want to express thanks to my church family for their steady support. Without their prayers, I would not have gone this far.

Finally, I would like to thank my dear family, my parents, my siblings, my husband Chun-Fa and my son Nathan. Their love and support gave me the strength to carry on in the most difficult time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Microprocessor performance has been growing at a rate of 60% per year in the past decade [68, 27]. The current generation, such as Alpha 21364 [25] and Intel Pentium 4 [23], is able to achieve over 1GHz clock rate. However, memory access time is increasing at a much slower rate, at about 7% per year [68, 27]. Current DRAM (Dynamic Ram) implementations usually take a few hundred naroseconds to retrieve data. As the performance gap between processors and memory continues to grow, techniques that reduce the effect of this disparity are essential to building 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 [60]. Caches are composed of SRAM (Static Ram) which has lower access latency but is more expensive compared to DRAM. The design of caches is based on one important program property - locality of references. Programs tend to reuse data that have been recently referenced. Caches are used to keep recently used data, thereby satisfying successive accesses to this data. Current memory systems usually adopt a three level memory hierarchy: the L1/L2 caches and DRAM main memory. The L1 cache is small, low-latency on-chip cache backed up by a larger, yet slower L2 cache. Only memory references that miss in both the L1 and L2 caches need to access main memory.

If programs exhibit good 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. For applications with regular access patterns several compiler techniques (e.g. blocking and loop transforms) [67] can be used to improve program locality. However, it is difficult for compilers to perform such optimization for applications with irregular access patterns. Thus techniques that hide memory latency are important in addition to the use of multi-level caches.

The idea of hiding memory latency is allowing CPU execution and memory accesses to proceed in parallel. One commonly used latency hiding technique is instruction scheduling [22], which tries to schedule load instructions ahead of dependent instructions so that data may arrive at the CPU in time. The freedom of instruction scheduling is limited by the available instruction level parallelism (i.e. independent instructions) in programs. When memory latency is large, it is usually hard to schedule load instructions early enough to effectively hide memory latency.

Another kind of latency hiding technique is prefetching. Prefetching predicts addresses of load instructions and issues memory requests earlier than demand fetches. Prefetching is speculative and non-faulting; that is, it does not incur any exception. Thus prefetching has more freedom to schedule memory requests ahead of dependent instructions. However, the effectiveness of a prefetching scheme depends on address prediction accuracy.

Prefetching for array-based numeric applications with regular access patterns has been well studied [30, 5, 11, 21, 59, 49, 8, 51, 34, 45]. However, prefetching for pointer-intensive applications remains a challenging problem. Conventional prefetching schemes rely on address stream regularity to predict future addresses. Many scientific computing applications, which use array data structures, present such regularity. Unfortunately, commercial applications, such as databases, graphics and VLSI applications, usually 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—called the pointer-chasing problem—since the address of the next node is not known until the contents of the current node are accessible. The pointer-chasing problem makes it difficult to schedule prefetch requests far enough ahead to actually hide memory latency.

In this thesis I present a novel prefetching framework, *the push architecture*, designed specifically for linked data structures (LDS). The push architecture resolves the address prediction problem by exploiting program structure (i.e. control flow and dependence relations among instructions) [55] instead of relying on address regularity. The push architecture identifies instructions that traverse LDS (LDS traversal kernels) statically and uses a micro-controller to execute traversal kernels only. This micro-controller is called a prefetch engine (PFE). A PFE can successfully generate future addresses when it runs ahead of the CPU.

To overcome the pointer-chasing problem, the push architecture applies a novel data movement model – *push*. The conventional architecture uses the pull approach, which initiates all memory requests (demand fetch or prefetch) from the processor or upper level of the memory hierarchy. I observe that, for pointer-based applications, performance will improve if lower levels of the memory hierarchy could dereference pointers and pro-actively *push* data up to the processor rather than requiring the processor to *fetch* each node before initiating the next request. 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.

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 pro-actively pushed up to the CPU. There are four main design issues in implementing the proposed push architecture:

1. Microarchitecture of the PFE

   The function of the PFE is to execute LDS traversal kernels. The PFE could be specialized hardware built for specific LDS traversal patterns or a programmable processor capable of executing a multitude of LDS traversal kernels. The latter approach has a flexibility advantage but may not perform as well as the former. I investigate both designs in this thesis.

2. 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 thesis, I 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.

3. 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.

4. 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 thesis, I 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 thesis 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, in-order programmable PFE), while the traditional pull method is not able to achieve any speedup. With throttling, the push architecture can achieve a performance comparable to that of a perfect memory system when there is enough computation between node accesses. Simulation also reveals that the push architecture retains its performance advantage over pull-based prefetching for future processors.

The push architecture is able to achieve significant speedups for most of the benchmarks tested using a simple, single-issue, in-order processor as the PFE. For list-based applications, the single-issue programmable PFE is able to deliver performance within 10% of that of the specialized hardware. Increasing the issue width of the programmable PFE can further improve performance. 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. I 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.

This thesis is organized as follows. Chapter 2 describes how prefetching hides memory latency and presents the challenges for prefetching linked data structures. Chapter 3 presents the high level overview of the push architecture. The implementation details are addressed in Chapter 4. In Chapter 5, I present simulation results demonstrating the effectiveness of the push architecture. Related work is discussed in Chapter 6. Chapter 7 concludes and presents possible extensions to this thesis. Appendix A lists traversal kernels of the applications studied in this thesis.

Chapter 2

Background

In this chapter, I first discuss several commonly used memory latency hiding techniques. I then describe how prefetching overlaps CPU execution with memory accesses and the fundamental issues in designing a prefetching scheme. This is followed by a discussion on the challenges in prefetching linked data structures.

2.1 Hiding Memory Latency

The essence of hiding memory latency is allowing CPU execution and memory accesses to proceed in parallel. To make such overlapping possible, a cache must continue to supply data to the CPU while processing misses, so the CPU can proceed as long as it is allowed by the data dependence. Such a cache is called a non-blocking cache [36]. To achieve more overlapping, compiler techniques, such as instruction scheduling [22] and software pipelining [37], reorder instructions to increase the distance (i.e., the number of instructions) between a load instruction and its uses. Modern superscalar processors, such as Alpha 21264 [33] and Intel Pentium processors [23], employ dynamic instruction scheduling to allow the processor to continue issuing instructions when dependences are present. The effectiveness of both static and dynamic approaches is limited by the available instruction level parallelism (i.e. independent instructions) in programs. When memory latency is large, it is difficult to find sufficient independent instructions to schedule so that memory latency can be fully overlapped with computation.

Another type of memory latency hiding technique takes advantage of thread level parallelism. Multi-threaded architectures, such as HEP [61], Tera [3], MASA [26] and April [1], offer fast context switches between threads. Memory latency is hidden by context switching to another thread once the running thread encounters a cache miss. Simultaneous Multithreading [65] improves the traditional multithreading architecture by simultaneously issuing instructions from different threads in the same cycle. These techniques increase processor utilization in a multiprogramming environment or improve performance for applications with inherent thread level parallelism, e.g. scientific computing and database applications. However, programs that are highly serial, such as pointer-based applications, can not benefit from this type of latency hiding technique.

Prefetching is another commonly used technique to hide memory latency. The push architecture proposed in this thesis is a prefetching framework designed specifically for linked data structures. Next, I describe how prefetching hides memory latency and the challenges in prefetching linked data structures.

2.2 Prefetching

Prefetching issues memory requests earlier than demand fetching a cache block when a cache miss occurs. Figure 2.1 illustrates how prefetching hides memory latency. Assume that a demand fetch q issuing at time t misses in both the L1 and L2 caches. So the data is only available to the CPU at time $t + mem\_latency$ as shown in Figure 2.1(a), where $mem\_latency$ is the time for retrieving data from main memory. If the available independent instructions in the program only allows the CPU to execute to time t', the CPU has to stall from t' to $t + mem\_latency$. Prefetching can reduce the CPU stall time by issuing memory request q earlier than time t. Furthermore, if q is issued no later than time

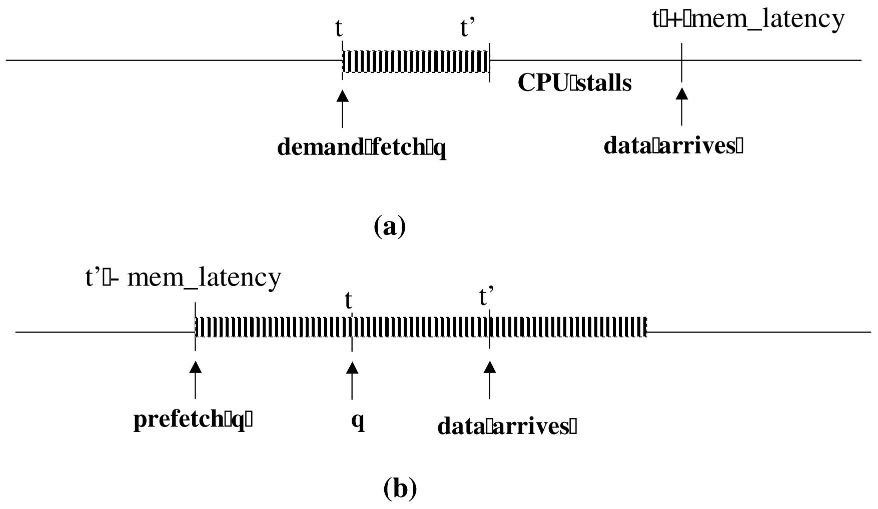


Figure 2.1: Illustration of Prefetching Process

$t' - mem\_latency$, the CPU stall time can be completely removed as shown in Figure 2.1(b).

Prefetching is speculative. It does not load values into registers and cannot cause an exception for virtual address faults and protection violation. This property allows prefetches to be issued at any point of program execution. Thus prefetching has more flexibility to schedule a memory request ahead of its use compared to techniques mentioned in the previous section.

To design a prefetching scheme, one needs to consider the following two issues:

1. Address prediction: the effective address of a load instruction is computed when it is issued. In order to issue prefetches before memory operations, we need an address prediction scheme to generate addresses before they are actually computed.

2. Prefetch schedule: timing is critical to the success of a prefetching scheme. If prefetches are issued too early, prefetched cache blocks may be replaced out of caches before they are accessed by the CPU. They are called early

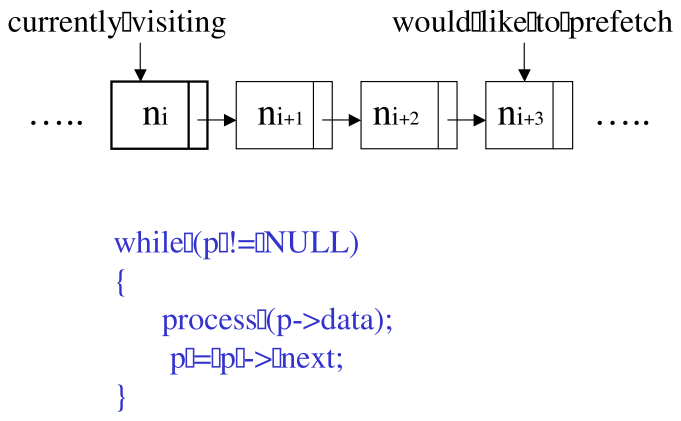
prefetches. If prefetches are issued too late, they can not effectively hide memory latency. Appropriate prefetch distance is determined by the amount of computation available for overlapping with memory accesses.

Prefetching for array-based applications has been well studied [30, 5, 11, 21, 59, 49, 8, 51, 34, 45]. However, prefetching for pointer-intensive applications is only explored in few studies [54, 56, 42] and remains a challenging problem.

2.3 Prefetching Array vs. Linked Data Structures

The fundamental difference between array and linked data structure (LDS) traversals is that the address streams from the former present arithmetic regularity while the latter do not. Consider traversing a linear array of integers versus traversing a linked-list. For linear arrays, the address of array elements accessed at iteration i is $A_{addr} + 4 * i$, where A_{addr} is the base address of array A and i is the iteration number. Using this formula, the address of element A[i] at any iteration i is easily computed. Furthermore, the above formula produces a compact representation of array accesses. Traditional prefetching schemes predict addresses relying on such arithmetic regularity. In contrast, LDS elements are allocated dynamically from the heap and adjacent elements are not necessarily contiguous in memory. Although linear layout of LDS may be achieved through careful data placement [14], the linearity property often disappears as the data structures evolve. Therefore, traditional array-based address prediction techniques are not always applicable to LDS accesses.

Without a compact address representation of LDS accesses, a series of pointer dereferences are required for generating future addresses. Consider the example



```
while (p != NULL)
{
     process (p->data);
      p = p -> next;
}
```

Figure 2.2: Illustration of the Pointer-Chasing Problem

shown in Figure 2.2. Assume that three nodes worth of computation is needed to hide memory latency. So we need to initiate a prefetch for node n_{i+3} while visiting node n_i in order to completely hide memory latency. However, to compute the address of node n_{i+3}, we must perform three pointer dereferences:

$$\text{p}\rightarrow\text{next},\ (\text{p}\rightarrow\text{next})\rightarrow\text{next},\ (\text{p}\rightarrow\text{next}\rightarrow\text{next})\rightarrow\text{next}.$$

This implies that we cannot prefetch node n_{i+3} until node n_{i+1} and n_{i+2} have been fetched. This is commonly called the pointer-chasing problem.

In summary, linked data structures present two challenges to any prefetching technique: address irregularity and the pointer-chasing problem. To improve the memory system performance of many pointer-based applications, both of these issues must be addressed.

Chapter 3

The Push Architecture

The push architecture is a cooperative software/hardware prefetching framework designed specifically for linked data structures. As mentioned in the previous chapter, one needs to consider two issues to design a prefetching scheme: address prediction and prefetch schedule. Prefetching linked data structures complicates these two issues because of address irregularity and the pointer-chasing problem. This chapter describes how the proposed push architecture overcomes these two challenges.

3.1 LDS Traversal Kernel

The push architecture exploits program structure for future address prediction instead of relying on address history [50, 54]. The idea is to separate memory accesses from computation by exploiting program structure and using a separate processor to execute memory accesses only. This separate processor is very likely to run ahead of the CPU and successfully generate future addresses because it does not perform computation on data.

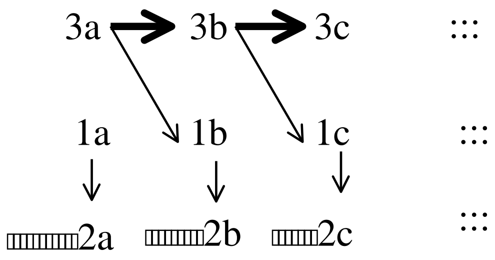
Consider the linked-list traversal example shown in Figure 3.1(a). The structure *list* is the LDS being traversed. Instruction 1 loads the address of structure x. Instruction 2 loads the data field of structure x. The program then performs computation on the data. Instruction 3 loads the address of the next node. By separating memory accesses from computation, we can obtain the LDS traversal kernel listed in Figure 3.1(b). The push architecture uses a micro-controller,

```
list = head;
while(list != NULL) {
  p = list->x;           (1)
  process (p->data);     (2)
  list = list->forward;  (3)
}
```

a) Source Code

```
while(list != NULL) {
  p = list->x;          /* (1) traversal load */
  p->data;              /* (2) data load */
  list = list->forward; /* (3) recurrent load */
}
```

b) LDS Traversal Kernel



c) Data Dependencies

Figure 3.1: Linked Data Structure (LDS) Traversal

called a prefetch engine (PFE), to execute LDS traversal kernels. The memory requests issued by the PFE access the memory hierarchy like demand fetches. Next, I describe important attributes of LDS traversal kernels and how they are constructed.

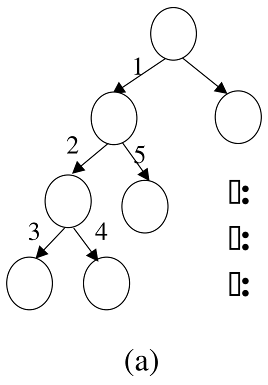
Attributes of LDS Traversal Kernels

Consecutive iterations of the traversal kernel create data dependencies as shown in Figure 3.1(c). Instruction 3 at iteration a produces an address consumed by itself and instruction 1 in the next iteration b. Instruction 1 produces an address consumed by instruction 2 in the same iteration. Loads in LDS traversal kernels can be classified into three types: *recurrent*, *traversal*, and *data* [54].

Instruction 3 (list=list→forward) generates the address of the next element of the *list* structure and is classified as a *recurrent* load. Instances of the recurrent load (3a,3b,3c,.. etc) form a dependence chain which is the critical path in the kernel loop as shown in Figure 3.1(c). Instruction 1 (p=list→x) produces addresses consumed by other pointer loads and is called a *traversal* load. Instruction 2 (p→data) is classified as a data load which accesses data other than memory addresses.

Constructing Traversal Kernels

The push architecture constructs LDS traversal kernels statically. A static approach has a larger analysis scope and can consider traversal orders (e.g. depth-first vs. breadth-first tree traversal) when constructing the kernels. Timing is crucial for the success of a prefetching scheme. With the information of LDS traversal orders, prefetches can be issued in the same order as demand fetches, thus likely resulting in timely prefetches. Roth et al. [54] propose identifying



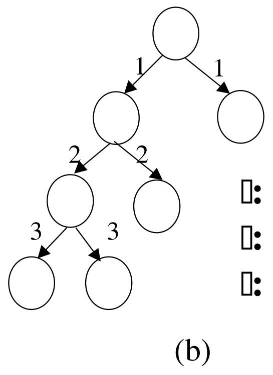


Figure 3.2: Example of Binary Tree Traversal in Depth-First Order

traversal kernels dynamically in hardware. Their method represents a 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. Consider traversing a binary tree in depth-first order as shown in Figure 3.2(a). Numbers marked at the tree edges represent the traversal sequence. Their approach prefetches all child nodes once the parent node is obtained. The prefetching sequence is shown in Figure 3.2(b). This may result in many early prefetches. For example, the recurrent load that leads to the right child node of the root is issued after the left subtree is traversed in the main CPU execution. But in the prefetching thread, it is issued at the beginning of tree traversal. Therefore, it could become an early prefetch because the right child node may be replaced out of the L1 cache before it is accessed by the CPU.

In this thesis, I construct LDS traversal kernels manually. I believe that the compiler techniques used by Luk et al. [42] can be extended to construct LDS traversal kernels automatically. In their work, they use type declaration information to recognize which data objects are LDS and control structure information

```
void *HashLookup(int key, hash hash)
{
    j = (hash->mapfunc)(key);
    for (ent = hash->array[j]; ent && ent->key != key; ent = ent->next);
    if (ent) return ent->entry;
    return Null;
}
```

(a) Source codes for Hash Table Lookup

```
void kernel (HashEntry ent, int key)
{
  for (ent ; ent && ent->key != key; ent = ent->next);
}
```

(b) Traversal kernel of Hash Table Lookup

Figure 3.3: Hash Table Lookup

to recognize when these objects are being traversed. However, the compiler study is beyond the scope of this thesis.

There are cases when traversal orders 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 as shown in Figure 3.3. For applications that determine the traversal order based on computation more than a value comparison, we can either aggressively prefetch all LDS elements or conservatively prefetch only those guaranteed to be accessed. The key thing is that the engine does not perform computation beyond simple value comparison for control flow.

Some applications dynamically change the data structure during traversal, such as bitonic sorting shown in Figure 3.4(a). This algorithm swaps the left and right pointers of a binary tree during traversal. If the traversal kernel traverses the whole binary tree, we will prefetch the wrong data after the CPU has modified

```
int Bimerge(root,spr_val,dir)
{
  pl = root->left; pr = root->right;
  :::
  while ((pl != Null)) {
    :::
    if (..) {
        :::
        SwapRight(pl,pr);
        :::
    } else {
        :::
        SwapLeft(pl,pr);
        :::
    }
    if (root->left != NULL) {
      root->value = bimerge(root->left,root->value,dir);
      spr_val = Bimerge(root->right,spr_val.dir);
    }
    :::
  }
}
```

(a) Bitonic Sorting

```
void kernel (root)
{
    left    = root->left;
    right   = root->right;
    data1   = root->value;
    ptr1    = left->left;
    ptr2    = left->right;
    data2   = left->value;
    ptr3    = right->left;
    ptr4    = right->right;
    data3   = right->value;
}
```

(b) Traversal Kernel

Figure 3.4: Bitonic Sorting

the structures. We can avoid prefetching down the wrong path by limiting the number of nodes accessed in the LDS traversal kernel. Figure 3.4(b) shows the traversal kernel which prefetches only the child nodes of the root. In this case, every node accessed by the CPU is treated as the root for the PFE.

3.2 The Push Model

To overcome the pointer-chasing problem, the push architecture employs a novel data movement model – *push*. The conventional data movement model initiates all memory requests (demand fetch or prefetch) from the processor or upper level of the memory hierarchy. It is called the pull model because cache blocks are moved up the memory hierarchy in response to requests issued from the upper levels. Figure 3.5(a) illustrates the data movement of the pull model. In

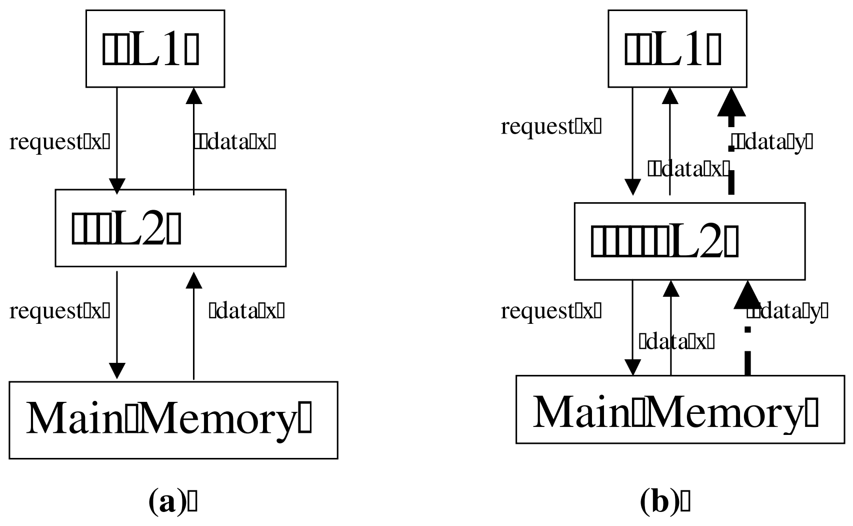


Figure 3.5: Pull vs. Push

contrast, the proposed push model allows data to be pushed up the memory hierarchy without a corresponding request issued from the upper levels as shown in Figure 3.5(b). This allows the implementation to perform pointer dereferences at the lower levels of the memory hierarchy and pro-actively push data up to the processor. The push model overcomes the pointer-chasing problem by decoupling the pointer dereference from the transfer of the current node up to the processor, which eliminates the request-response delay required in the conventional pull model.

In the pull model, the prefetch process is serial because pointer dereferences are required to generate addresses for successive prefetch requests (recurrent loads). Consider the prefetch schedule shown in Figure 3.6(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 $i+1$

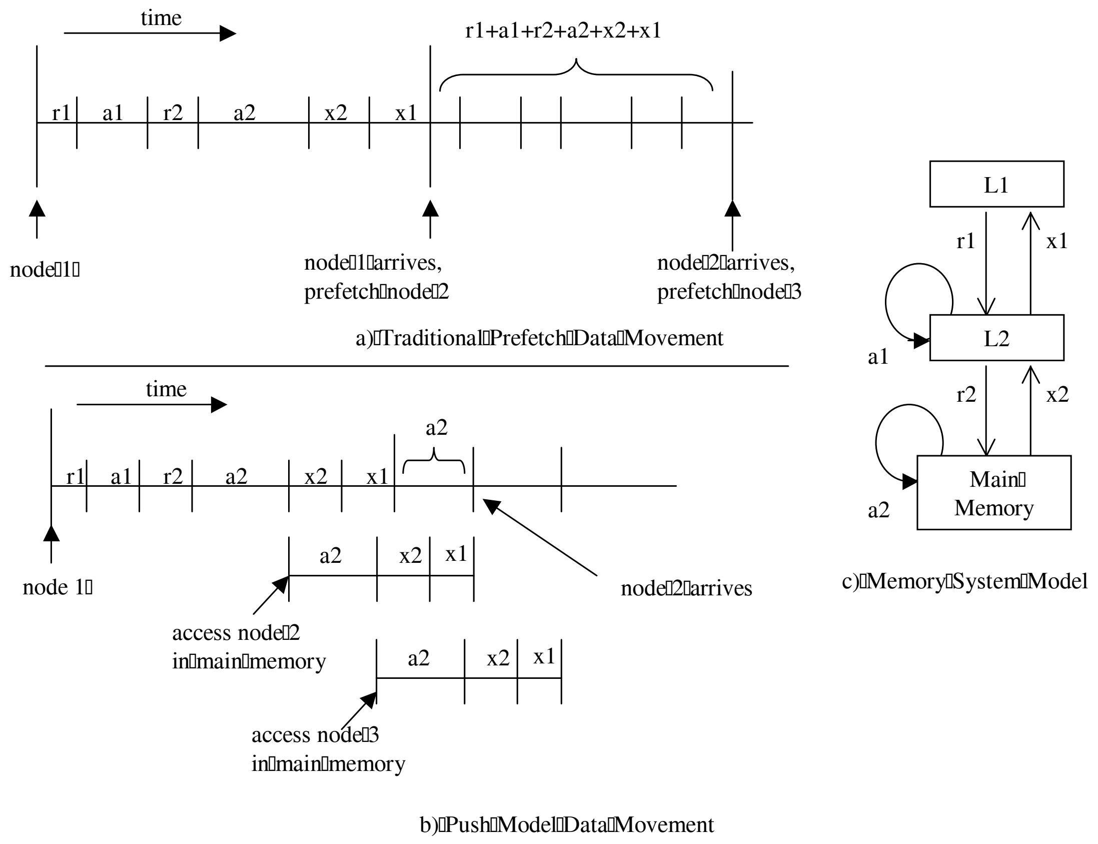

time
r1+a1+r2+a2+x2+x1
r1
a1
r2
a2
x2
x1
node 1
node 1 arrives, prefetch node 2
node 2 arrives, prefetch node 3
a) Traditional Prefetch Data Movement
time
r1
a1
r2
a2
x2
x1
a2
node 1
access node 2 in main memory
access node 3 in main memory
node 2 arrives
b) Push Model Data Movement
L1
r1
x1
L2
a1
r2
x2
Main Memory
a2
c) Memory System Model

Figure 3.6: Data Movement for Linked Data Structures

arrives at the CPU $(r1 + a1 + r2 + a2 + x2 + x1)$ cycles (the round-trip memory latency) after node i.

In the push model, pointer dereferences are performed at the lower levels of the memory hierarchy. This eliminates the request from the upper levels to the lower level, and allows us to overlap 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 3.6(b) (Note: in this simple model, I ignore the possible overhead for address translation. Section 5.3.8 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 from $(r1 + a1 + r2 + a2 + x2 + x1)$ to $a2$. In the Alpha 21264 architecture [33], the DRAM access time is 48 cycles and round-trip memory latency is 84 cycles. So the total latency is potentially reduced by 43%.

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 3.7 illustrates how C affects prefetching performance for both the push and pull model in the following three scenarios: (1) $C >= (r1 + a1 + r2 + a2 + x1 + x2)$ (2) $a2 <= C < (r1 + a1 + r2 + a2 + x1 + x2)$ and (3) $C < a2$. Below I discuss each of these in more detail.

1. $C >= (r1 + a1 + r2 + a2 + x1 + x2)$:

   Figure 3.7(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

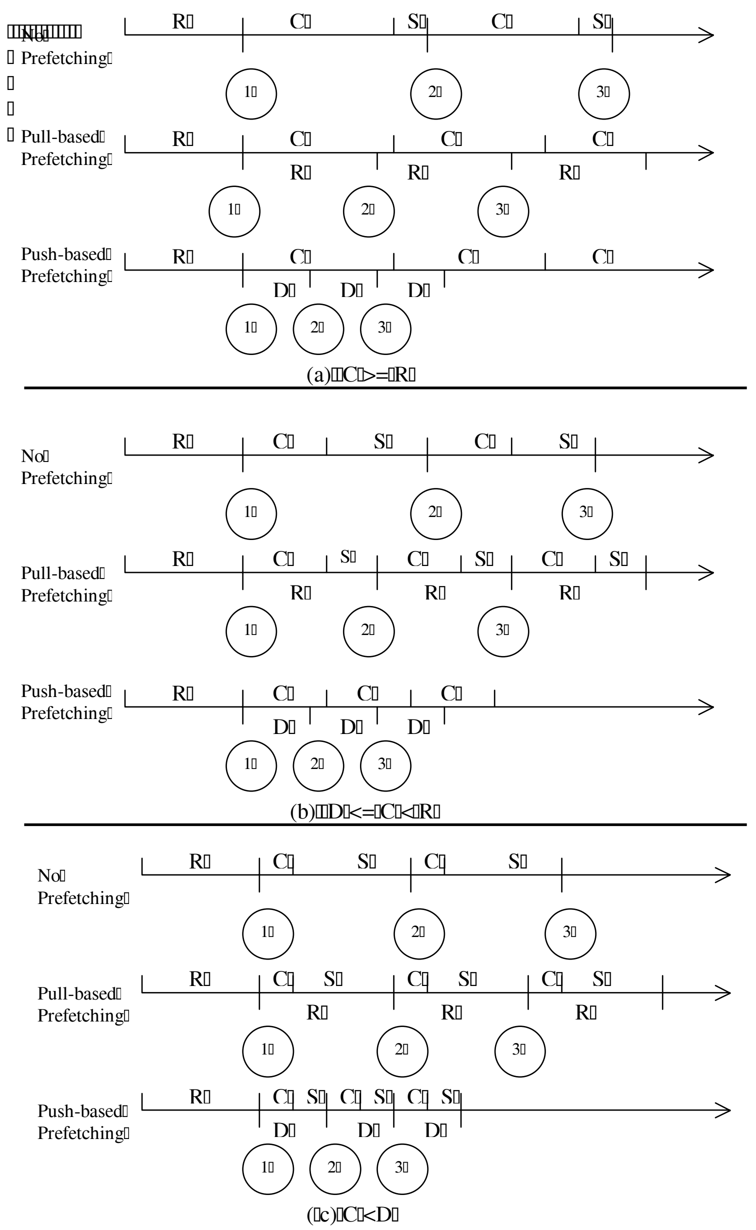


Figure 3.7: Prefetching Performance: Push vs. Pull. R=round-trip memory latency; D=DRAM access time; C=computation time; S=memory stall time

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. $a2 <= C < (r1 + a1 + r2 + a2 + x1 + x2)$:

   Figure 3.7(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 3.7(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. I evaluate the push model performance using a set of pointer-based applications in Chapter 5.

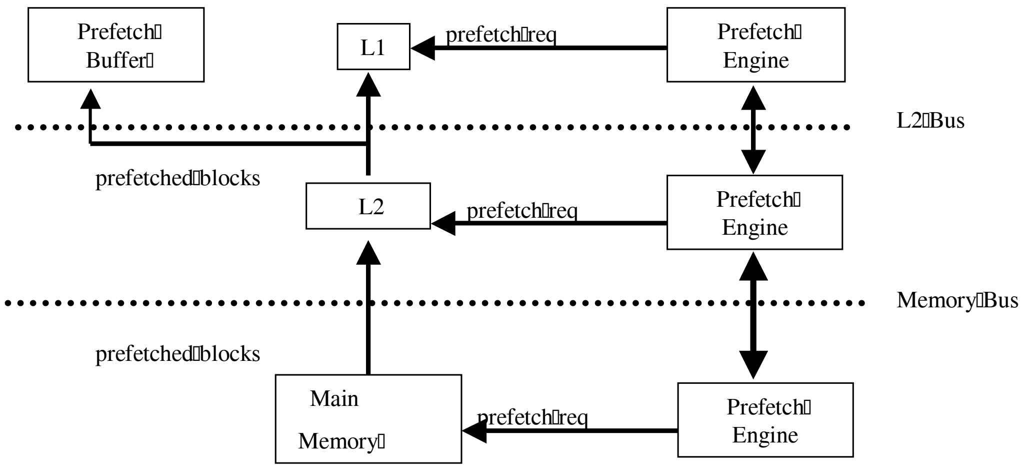


Figure 3.8: The Push Architecture

3.3 Design Issues of the Push Architecture

To realize the push model, the push architecture attaches a prefetch engine to each level of the memory hierarchy. Figure 3.8 shows the block diagram of the push architecture. 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. If the LDS node size is larger than the cache block size, only blocks accessed by the PFEs are prefetched. The prefetch buffer is a small, fully-associative cache, which can be accessed in parallel with the L1 cache.

The push architecture attaches the PFE to each level of the memory hierarchy so that pointer dereferences can occur at the level where LDS elements reside. The L1 PFE is first activated to execute the traversal kernel. The lower level PFEs are triggered once a recurrent load issued from the L1 PFE misses in the L1 cache. If this recurrent load is a hit in the L2 cache, the L2 PFE is activated. Otherwise, on a L2 miss, the memory PFE begins prefetching. The details of the interaction scheme among three PFEs are described in Section 4.2.

To implement this push architecture, we first need to define the microarchitecture of the PFE, which determines how fast prefetches can be issued. The push architecture also needs to employ an adaptive mechanism to adjust prefetch distance according to a program's run time behavior. Techniques to avoid redundant and useless prefetches are also important for prefetching performance. Below I discuss design issues that need to be addressed to implement the push architecture.

1. Microarchitecture of the PFE

   The function of the PFE is to execute LDS traversal kernels. The PFE architecture determines how efficient the PFE can issue prefetches. The PFE could be specialized hardware built for specific LDS traversal patterns. This approach could achieve high prefetching performance but it is difficult to have one optimized design for all types of LDS traversal patterns. The other approach uses a general purpose processor core as the PFE so it can execute a multitude of LDS traversal kernels. The programmable PFE design has a flexibility advantage but it may not perform as well as the specialized PFE. In this thesis, I explore both design approaches (see Section 4.1).

2. Synchronization between the CPU and PFE execution

   The PFE executes traversal kernels independent of the processor execution. However, synchronization between the CPU and PFE execution is necessary when the PFEs run too far ahead of the processor or generate useless prefetches, e.g. mis-predicted prefetches. In this thesis, I design an adaptive mechanism that throttles the PFE execution according to a program's runtime behavior (see Section 4.3).

3. Redundant prefetches

One potential problem for the push model is that the cache blocks pushed up by the lower level engines could already exist in the upper level because of program locality. For example, the second visit to a node in the depth-first tree traversal could become a redundant prefetch. Redundant prefetches could affect performance by wasting bus bandwidth or causing the PFE to run behind the main computation since memory accesses are satisfied more slowly in the lower levels of the memory hierarchy.

To reduce the performance impact of redundant prefetches, I use a small data cache in the L2 and memory PFE to filter out redundant prefetches. The PFE data cache captures recently accessed cache blocks by the PFE. Only prefetches that miss in the PFE data cache access the L2 cache and main memory (see Section 4.4).

4. Address translation

   Since addresses generated from the PFE execution are virtual addresses, a prefetch request needs to go through address translation before being sent to the cache/memory controller. The CPU uses a TLB (Translation Look-aside Buffer) to perform fast virtual to physical address translation. The push architecture uses the same technique to perform address translation for prefetch requests. I place a TLB in the L2 and memory PFEs. Since the L1 PFE is close to the CPU, it shares one TLB with the CPU. The push architecture implements a hardware TLB miss handler so a TLB miss incurred by a prefetch does not interrupt the CPU execution. TLBs at different levels of the memory hierarchy are updated independently.

   A TLB entry becomes invalid once the corresponding page is replaced from main memory because of a page fault. The operating system is responsible for updating the CPU TLB to ensure program correctness. The same TLB entry could also exist in the PFE TLBs. We can update the PFE TLBs as well but it requires extra support from the operating system to maintain coherency among these TLBs. Since the PFE execution does not affect program correctness, we can choose not to update the PFE TLBs. This only results in mis-predicted prefetches and does not affect program correctness.

5. Context switch

   In a 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 re-activate 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 thesis focuses on the performance impact of the push model on a single program, so I do not evaluate how a context switch affects prefetching performance further.

6. Number of PFEs

   The push architecture described above attaches the PFE to each level of the memory hierarchy (3_PFE). To reduce the hardware cost, I present two

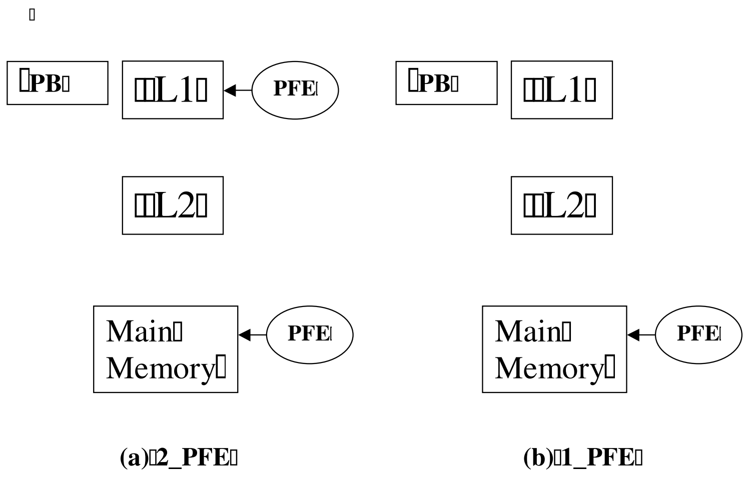


Figure 3.9: Variations of the Push Architecture

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 3.9 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 3.2, 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 [58, 70]. 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. Section 5.3.6 evaluates the performance of these architectures.

7. Speed of PFEs

   The success of the push architecture requires the PFEs to run enough ahead of the main execution. Therefore, the relative speed between the PFE and the CPU could have a significant impact on prefetching performance. If only considering performance, we want the PFEs to run at least the same clock rate as the CPU so they are less likely to run behind the main execution. However, it might be costly to build such a system. Generally, load instructions issued from the memory PFE are resolved in the DRAM access time. So the speed of DRAM has more effect on how fast prefetches are issued from the main memory level than that of the memory PFE. Section 5.3.7 evaluates how different memory PFE clock rates affect the push model performance.

In the next chapter, I 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.

Chapter 4

Implementation of The Push Architecture

In this chapter, I describe the implementation details of the push architecture. Section 4.1 describes the architecture of the prefetch engine. Section 4.2 explains how the PFEs at different levels of the memory hierarchy interact with one another. Then, I present a mechanism to synchronize the CPU and PFE execution in Section 4.3. Section 4.4 describes a technique to reduce the performance impact of redundant prefetches. The push model also places new demands on the memory hierarchy. Section 4.5 examines this issue.

4.1 PFE Design

The task of the PFE is to execute traversal kernels. There are two approaches for the PFE design. One is the specialized PFE design using an application specific integrated circuit (ASIC). The other approach adopts a commodity processor core as the PFE. The specialized PFE has a performance advantage but it is difficult to have one optimized implementation for all types of LDS traversal patterns. Using a general purpose processor as the PFE may not be able to achieve the same performance as the specialized PFE, but it can support a multitude of LDS traversal kernels. In this thesis, I explore both approaches. I design the specialized PFE for linked-list traversal. I also present the programmable PFE design using a simple in-order general purpose processor.

```
While (list != NULL) {
  p = list->p;
  process(p->f1);
  process(p->f2);
  process(p->f3);
  process(p->f4);
  list = list->next;
}
```

(a) C Sources

```
While (list != NULL) {
  p = list->p;
  d1 = p->f1;
  d2 = p->f2;
  d3 = p->f3;
  d4 = p->f4;
  list = list->next;
}
```

(b) Traversal Kernel

```
          beq $24, null, $exit
$L1: 1.   ld  $20, 20($24)
     2.   ld  $3,  4($20)
     3.   ld  $4,  8($20)
     4.   ld  $5,  12($20)
     5.   ld  $6,  16($20)
     6.   ld  $24, 32($24)
          bne $24, null, $L1
$exit:
```

(c) Traversal Kernel in Assembly

$$\begin{tikzcd}
6a \arrow[r] \arrow[dr] & \underset{([6a]+32)}{6b} \arrow[r] \arrow[dr] & \underset{([6b]+32)}{6c} & ::: \\
& \underset{([6a]+20)}{1b} \arrow[d] & \underset{([6b]+20)}{1c} \arrow[d] & ::: \\
& :::: & ::::
\end{tikzcd}$$

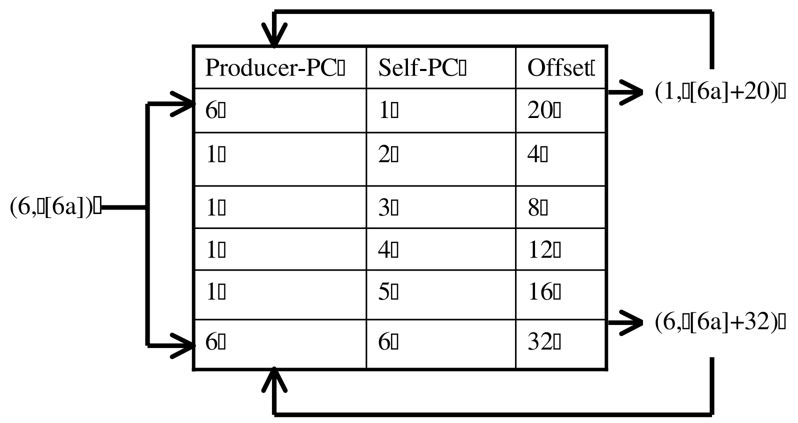
(d) Address Generation of LDS Kernel Execution

Figure 4.1: Linked-List Traversal Example

4.1.1 Specialized PFE Design

Consider the linked-list traversal example in Figure 4.1. The specialized PFE is designed to execute iterations of kernel loops until the returned value of the recurrent load (instruction 6: list=list→next) is NULL. Instances of the recurrent load (6a, 6b, 6c,.. etc) form a dependence chain that is the critical path in the kernel loop, as mentioned in Section 3.1. Therefore, an optimized PFE design must be able to move ahead along this critical path as fast as possible. For example, the PFE should be able to issue 6c once the result of 6b is ready even though 1b is not resolved yet; that is, an optimized PFE design should support out-of-order execution.

The specialized PFE design exploits one particular property of the address



Producer-PC	Self-PC	Offset
6	1	20
1	2	4
1	3	8
1	4	12
1	5	16
6	6	32

Figure 4.2: Traversal Kernel Table (1)

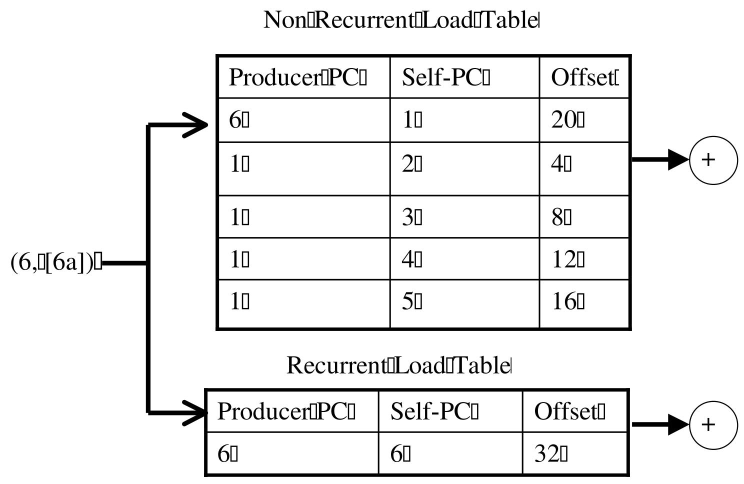
generation of LDS traversals. Recall that load instances in LDS traversal kernels present data dependences which can be characterized as producer and consumer relation. The result of a producer is used as the base address of its consumer. Figure 4.1(d) illustrates the address generation process of the linked-list traversal example. For instance, the address of 6b is [6a]+32, where [6a] is the result of 6a. The design concept of the specialized PFE is to represent LDS traversal kernels in the form of producer and consumer pairs and use the completion of a load instruction to drive execution [54]. The following discussion uses the linked-list traversal kernel in Figure 4.1 as a running example.

A load instruction in a traversal kernel is first transformed into the following format:

1. Producer_PC: the Program Counter (PC) of the instruction that produces the base address of this load.

2. Self_PC: this load instruction's PC

3. Offset: this load instruction's offset



Non Recurrent Load Table

Producer PC	Self-PC	Offset
6	1	20
1	2	4
1	3	8
1	4	12
1	5	16

Recurrent Load Table

Producer PC	Self-PC	Offset
6	6	32

Figure 4.3: Traversal Kernel Table (2)

For example, instruction 1 is transformed into (6, 1, 20). A traversal kernel is stored in a table using Content Addressable Memory as shown in Figure 4.2. This table is indexed using the Producer PC. When a load instruction completes, its PC is used to search this table if the result is not NULL. For each matching entry, the effective address of the dependent instruction is generated by adding its offset to the result of the probing load. For example, when 6a completes, 6 is used to search the traversal kernel table. This results in two matching entries (i.e. instruction 1 and 6). The addresses of these two dependent loads are [6a]+20 and [6a]+32, respectively. When these two loads complete, their PCs are again used to search this table. In this way, the PFE can continue to issue memory requests until the result of the recurrent load is NULL. This design achieves out-of-order execution effect because a load instruction can be issued even though loads that proceed in program order are not resolved yet.

To further optimize the design, we would like to give a recurrent load priority if there are multiple loads ready in the same cycle. To achieve this optimization,

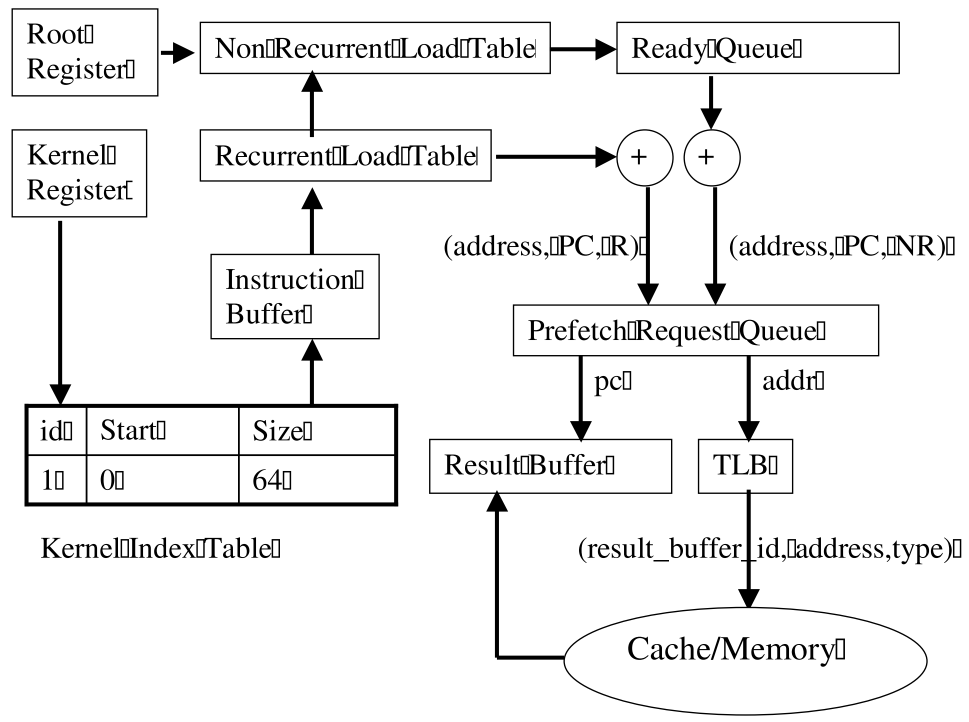


Figure 4.4: Block Diagram of the Specialized PFE

the PFE executes a traversal kernel in two separate threads by storing a traversal kernel in two tables as shown in Figure 4.3. The Recurrent Load Table stores only recurrent loads and the Non-Recurrent Table stores other loads. When a load completes, the PFE probes both tables using its PC. The PFE uses two adders so address calculation of the two threads can proceed in parallel.

The complete PFE block diagram is shown in Figure 4.4. The Instruction Buffer stores traversal kernels, which are downloaded to the PFE through a memory-mapped interface at the beginning of program execution. Since there could be more than one traversal kernel in a program, each kernel is associated with a unique identifier. During program execution, only kernel identifiers need to be conveyed to the PFEs. In this way, the overhead of fetching traversal ker-

```
void *HashLookup(int key, hash hash)
{
   j = (hash->mapfunc)(key);
  write_kernel_id(1);
  write_root_reg(hash->array[j]);
   for (ent = hash->array[j]; ent && ent->key != key; ent = ent->next);
   if (ent) return ent->entry;
   return Null;
}
```

Figure 4.5: Source of the Hash Table Lookup

nels only occurs once. The Kernel Index Table stores the information for locating a traversal kernel in the Instruction Buffer using its kernel identifier.

Operations of the Specialized PFE

Before a program starts traversing a LDS, the root address of the LDS being traversed and the traversal kernel identifier are written into the Root and Kernel registers. Both registers are memory-mapped. Figure 4.5 uses a hash table lookup example to show how a program conveys these two pieces of information to the PFE. The *for* loop in the example traverses a list. Hash$\rightarrow$array[j] is the root address of the list being traversed, and the kernel identifier is 1. Before the *for* loop, these informations are passed to the PFEs through store operations. Note that the root address is only written to the Root register of the L1 PFE, but the kernel identifier should be written into the Kernel registers of all PFEs. The overhead of these instructions is low because they can be scheduled in unused issue slots in a superscalar processor.

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 Recurrent and Non-Recurrent Load Table. In order to distinguish a recurrent load from others, the first instruction of a traversal kernel is always a recurrent load. Since a linked-list traversal kernel has one recurrent load, only the first instruction is loaded into the Recurrent Load Table. A write to the Root register activates the PFE, which starts execution by probing these two tables using the PC of a recurrent load and using the Root register value as the initial base address.

Since there could be more than one matching entry when searching the Non-Recurrent Load Table, the Ready Queue stores searching results, containing a load's PC, base address (i.e. the result of the probing load) and offset. At each cycle, the first entry of the Ready Queue is processed, and a new prefetch request is formed by adding its offset to the base address. A prefetch request is then stored in the Prefetch Request Queue along with its PC and type (R: recurrent prefetch; NR: non-recurrent prefetch).

The PFE contains a TLB for address translation. A prefetch request goes through virtual to physical address translation before being sent to the cache or main memory. The Result Buffer stores the information required for the PFE to continue execution once a load is resolved. Before a prefetch request is issued, an entry of the Result Buffer is allocated to store its PC. A prefetch request is then issued to the cache/memory along with its type and Result Buffer identifier. I explain why the cache/memory controller needs to be aware of the request type in Section 4.2.

The specialized PFE presented above is a highly optimized architecture for linked list traversals. It achieves out-of-order execution and gives recurrent load priorities. In the next section, I describe a PFE design that is capable of executing different LDS traversal kernels.

4.1.2 Programmable PFE Design

To design a programmable PFE I adopt a general purpose processor core. The PFE implements a 5 stage pipeline:

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.

Since the task of the PFE is to execute traversal kernels, 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. Furthermore, since LDS traversal kernels are highly serial, an in-order processor should be able to provide satisfactory performance. I evaluate how issue width affects prefetching performance in the next chapter.

The PFE supports three types of load instructions. They are *ld_recurrent*, *ld_data* and *ld_local*. *Ld_recurrent* is used for recurrent loads. The *ld_data* instruction is a non-binding load similar to the existing prefetch instructions. It is used for loads that do not have dependent instructions. The third load type 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. The height of

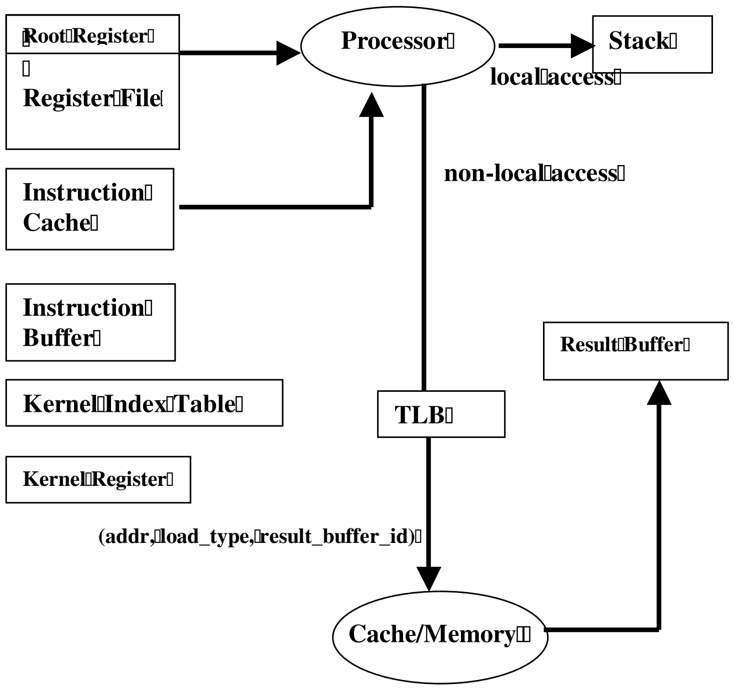


Figure 4.6: Block Diagram of the Programmable PFE

the largest binary tree (8 byte tree node) that fits in a 512 MB physical memory is 24. The maximum stack size for a 24-level tree is less than 4K, assuming all registers (32 integer registers) are modified in the program. So one cycle access time is feasible. Since the height of a tree grows logarithmically with the memory size, one cycle access time to the PFE stack should still be feasible for future computer systems.

The programmable PFE structure is shown in Figure 4.6. The Instruction Buffer, Root/Kernel register and Kernel Index Table function the same as in the specialized PFE. When the current traversal kernel identifier is written into the Kernel register, the corresponding traversal kernel is loaded into 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. A memory request from the ld_local instruction is sent to the Stack. Other memory requests are sent to the cache/memory controller after address translation.

4.2 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, I focus on how three PFEs work together to perform prefetching. Next section presents the complete interaction scheme among the CPU and PFEs.

When to Activate the Lower Level PFEs?

The L1 PFE is activated when the root address of the LDS being traversed is written into the Root register. The L1 PFE continues execution until a recurrent load 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 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, I use its location to decide which PFE should perform prefetching. When a recurrent load causes a L1 miss, the L1 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. Recall that a prefetch request is sent to the cache controller along with its type: recurrent or non-recurrent. So the controller can distinguish a recurrent prefetch by simply checking its type.

Other load instructions in the traversal kernel may access the same cache block as a recurrent load. Figure 4.7(a) shows a traversal kernel example which has five load instructions. The first load instruction (tree→color) accesses the color field of the tree structure. The other four instructions are recurrent loads (tree→nw, tree→ne, tree→sw, tree→se). Since the PFE is an in-order issue processor, if tree→color and tree→nw access the same cache clock, tree→nw is always a cache hit. As a result, the lower level PFEs are never activated. To solve this problem , I reorder the traversal kernel such that the first load instruction is always a recurrent load as shown in Figure 4.7(b). The potential problem of reordering the traversal kernel is that we may trigger the lower level PFEs to traverse down the wrong path. In Section 4.3, I describe a mechanism that can re-activate the PFEs to traverse on the correct path when this scenario occurs.

How to Resume Execution of the Upper Level PFEs?

The upper level PFEs may need to resume execution again after they stop prefetching due to a recurrent load miss. Consider traversing a binary tree in depth-first order, as shown in Figure 4.8. The assembly code is the traversal kernel. Assume that the recurrent load x (in Figure 4.8(a)), between node 1 and 2 in Figure 4.8(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

```
void perimeter_kernel(QuadTree tree)
{
    QuadTree nw;

    if (tree->color==grey)
    {
        perimeter_kernel(tree->nw);
        perimeter_kernel(tree->ne);
        perimeter_kernel(tree->sw);
        perimeter_kernel(tree->se);
    }
}
```

(a) Traversal kernel before reordering

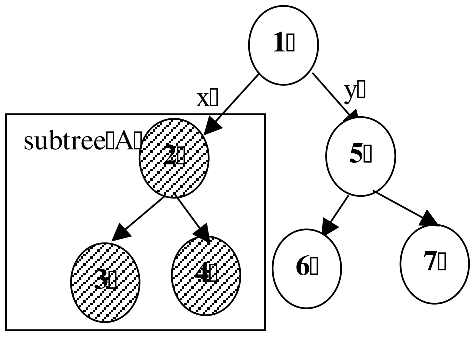
```
void perimeter_kernel(QuadTree tree)
{
    QuadTree nw;

    if (!tree) return;

    nw = tree->new;

    if (tree->color==grey)
    {
        perimeter_kernel(nw);
        perimeter_kernel(tree->ne);
        perimeter_kernel(tree->sw);
        perimeter_kernel(tree->se);
    }
}
```

(b) Traversal kernel after reordering

Figure 4.7: Traversal Kernel Example

```
        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

Figure 4.8: Tree Traversal Example

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.

I 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 4.8(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 4.8, the code segment from 0x400958 to 0x400968 is responsible for saving register values to the stack. When the engine

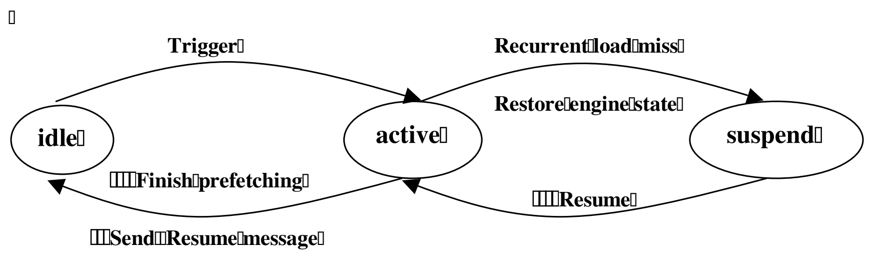


Figure 4.9: State Diagram of the Interaction Scheme

is signaled to stop execution because the recurrent load x misses in the L1 cache (which corresponds to instruction 0x400988), it can be executing any instruction between 0x400950 and 0x400978 (the PFE will 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 need 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 masking interrupts to achieve atomic execution.

To resume execution, I 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 will keep the token and resume execution. From the previous example, when the memory engine finishes prefetching subtree A, it sends a Resume token up the memory hierarchy. The L2 PFE sees this token first. Since it is not in the suspend state, the Resume token continues to travel up to the L1 level. When the L1 engine sees this token, it resumes execution.

The state diagram shown in Figure 4.9 summarizes the interaction among the PFEs. When the Root register is written, a PFE transits to the active state. Once a recurrent load miss occurs, the engine suspends execution, restores the correct state and moves to the suspend state. When a PFE finishes executing the traversal kernel, it sends a Resume token on the PFE Channel and returns to the idle state. A PFE in the suspend state resumes execution when it sees a Resume token.

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

4.3 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 a program's runtime behavior.

The idea of this scheme is to throttle PFE execution to match the rate of processor data consumption. The throttle mechanism is built around the prefetch buffer, where the PFEs produce its contents and the CPU consumes them. I

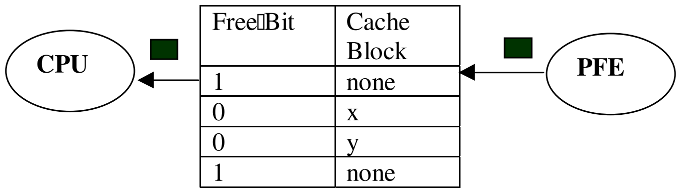


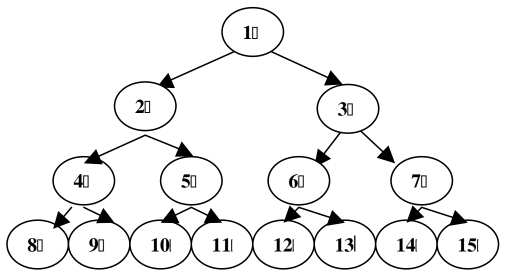
Figure 4.10: Prefetch Buffer Diagram

augment each cache line of the prefetch buffer with a free bit [61] as shown in Figure 4.10. The free bit is set to 0 when a new cache block is brought in. Once the processor accesses this block, 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 contains free entries, it implies that the PFEs are performing useful prefetches because the CPU is consuming prefetched cache blocks regularly. 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. This is similar to a commonly used link-level flow control mechanism, where the free space in the destination input buffer is used to control the input rates [17].

If the prefetch buffer is full, 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.

Figure 4.11 illustrates these three scenarios assuming that the size of a LDS element is equal to the cache block size and the prefetch buffer can contain three cache lines. Figure 4.11(a) shows the first scenario where the PFE is running too



Time: 0 t1 t2 t3 t4 t5

PB:

Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data
1	-	0	1	0	2	0	2	0	2	1	2
1	-	1	-	1	-	0	4	0	4	0	4
1	-	1	-	1	-	1	-	0	8	0	8

CPU : node 1 ——— processing node 1 ———→ node 2

(a) The CPU is behind the PFE execution

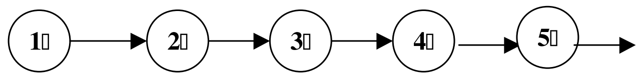
Time: 0 t1 t2 t3 t4

PB:

Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data
1	-	0	1	0	2	0	2	0	2
1	-	1	-	1	-	0	4	0	4
1	-	1	-	1	-	1	-	0	8

CPU : node 1 ——— node3 ——— node 6 ⟶

(b) The PFE is traversing down the wrong path



Time: 0 t2 t3 t4

Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data	Free Bit	Data
1	-	1	-	0	2	0	2	0	2
1	-	1	-	1	-	0	3	0	3
1	-	1	-	1	-	1	-	0	4

CPU: node 1 ——— node 2 ——— node 3 ——— node 4 ——— node 5 ⟶

(c) The PFE is behind the CPU execution

Figure 4.11: Synchronization between the CPU and PFE Execution

far ahead of the CPU. The CPU traverses a binary tree in depth-first order. At time t4, the CPU is still processing node 1, but the PFE has run ahead of the CPU and filled up the prefetch buffer. The prefetch buffer remains full until the CPU accesses node 2 at time t5. During this period (t4 to t5), we would like to suspend the PFE execution so the PFEs do not run too far ahead of the CPU.

Figure 4.11(b) illustrates the second scenario where the PFE is traversing down the wrong path. Assume that the CPU accesses only tree node 1, 3, 6 and 12 but the PFE traverses the entire tree in depth-first order. So at time t4, the PFE has filled up the prefetch buffer with node 2, 4, and 8, which the CPU will never access in the future.

Figure 4.11(c) uses a list traversal example to demonstrate the third scenario. Let's assume that node 1 misses in both the L1 and L2 cache, so the memory PFE is activated to prefetch node 2, 3,.. etc. If node 2, 3 and 4 happen to hit in the L1 cache, the memory PFE may run behind the CPU since a load instruction is resolved more slowly in the memory PFE than in the CPU. At time t4, the PFE fills up the prefetch buffer with node 2, 3 and 4, but the CPU has already traversed up to node 5. Similar to the second case, the CPU will not consume the data in the prefetch buffer in the future.

When the prefetch buffer is full, the PFEs should suspend execution in all three cases. This allows the CPU to catch up the PFE execution in the first case and prevents the PFEs from performing useless prefetches in the other two cases. To achieve this, the L1 cache controller monitors the state of the prefetch buffer. Once the prefetch buffer is full, it 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 the prefetch buffer has free entries, the L1 cache controller should broadcast a Pb_Free message to notify the PFEs to resumes execution. To prevent the prefetch buffer from constantly switching between the full and free state, the L1 cache controller does not send a Pb_Free message until there are at least two free entries in the prefetch buffer. 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, the PFEs should start a new prefetching thread because the current execution is not producing useful prefetches. Here, I use a recurrent load miss from the CPU execution as a synchronization point. When the prefetch buffer is full and a recurrent load miss occurs, it is a hint that the engine is behind or running on the wrong path. Note that this scenario could still occur even though the PFEs are actually running ahead of the CPU because a prefetched cache block is replaced out of the L1 cache before subsequent accesses. Therefore, the L1 cache controller only notifies the PFEs to stop execution (i.e. broadcasting a Stop message) after observing two consecutive recurrent load misses. The PFEs return to the idle state once observing a Stop message. The second recurrent load miss then serves as a triggering point by setting its type to R. As described in Section 4.2, the PFE at the level where this miss is satisfied will be activated. To easily distinguish a recurrent load miss, I add a load variant–ld_recurrent– to the CPU instruction set.

If the PFEs are triggered using the result of a recurrent load instead of the root address of the entire LDS, they can only prefetch a subset of the LDS for tree traversal. Consider the tree traversal example shown in Figure 4.12. Assume that the recurrent load x misses in the L1 cache and prefetch buffer. If the prefetch buffer is full when this miss occurs, based on the synchronization mechanism

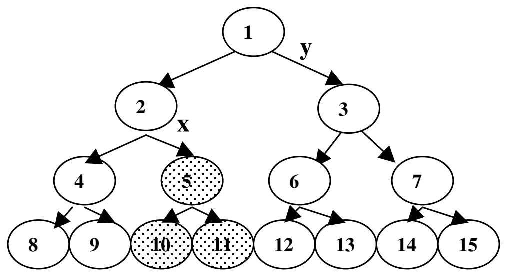


Figure 4.12: Tree Traversal Example

described above, the L1 cache controller notifies all the PFEs to return to the idle state, and the PFEs start a new prefetching thread using the result of x as the root address. So this new thread can only prefetch a subset of the tree (i.e. node 5, 10 and 11). To allow the PFEs to prefetch the remaining tree, a recurrent load miss from the CPU execution serves as a triggering point if all the PFEs are in the idle state when this miss occurs. For example, assume that the recurrent load y, between node 1 and 3, misses in both the L1 cache and prefetch buffer. If the PFEs have finished prefetching node 5, 10 and 11 at this point (i.e. all the PFEs are in the idle state), we can activate the PFEs again by setting y's type to R. This scheme requires the PFE to notify the L1 cache controller when it starts and finishes prefetching.

The complete PFE state diagram including the throttle mechanism is shown in Figure 4.13. Table 4.1 lists all event types and how these events occur. A PFE transits to the active state when its Root register is written. When a recurrent load miss occurs, an engine suspends execution, restores the correct engine state and moves to the suspend state. When a PFE finishes executing the traversal kernel, it sends a Resume message up the memory hierarchy through the PFE Channel and returns to the idle state. A PFE in the suspend state

Event Type	Description
Stop	1. When the CPU starts a new LDS traversal, it broadcasts a Stop message through the PFE channel. 2. When the prefetch buffer is full and two consecutive recurrent load misses from the CPU execution occur, the L1 cache controller broadcasts a Stop message through the PFE channel.
Trigger	The cache/memory controller writes the value of a recurrent load to the Root register of the PFE.
Recurrent_Load _Miss	When a recurrent prefetch misses in the cache, the cache controller signals the PFE to suspend execution.
Pb_Full	The L1 cache controller broadcasts a Pb_Full message through the PFE channel once it detects that the prefetch buffer is full.
Pb_Free	The L1 cache controller broadcasts a Pb_Free message through the PFE channel once the prefetch buffer has at least two free entries .
Resume	When the PFE finishes executing traversal kernels, it sends a Resume message up the memory hierarchy through the PFE channel.

Table 4.1: PFE State Event Types

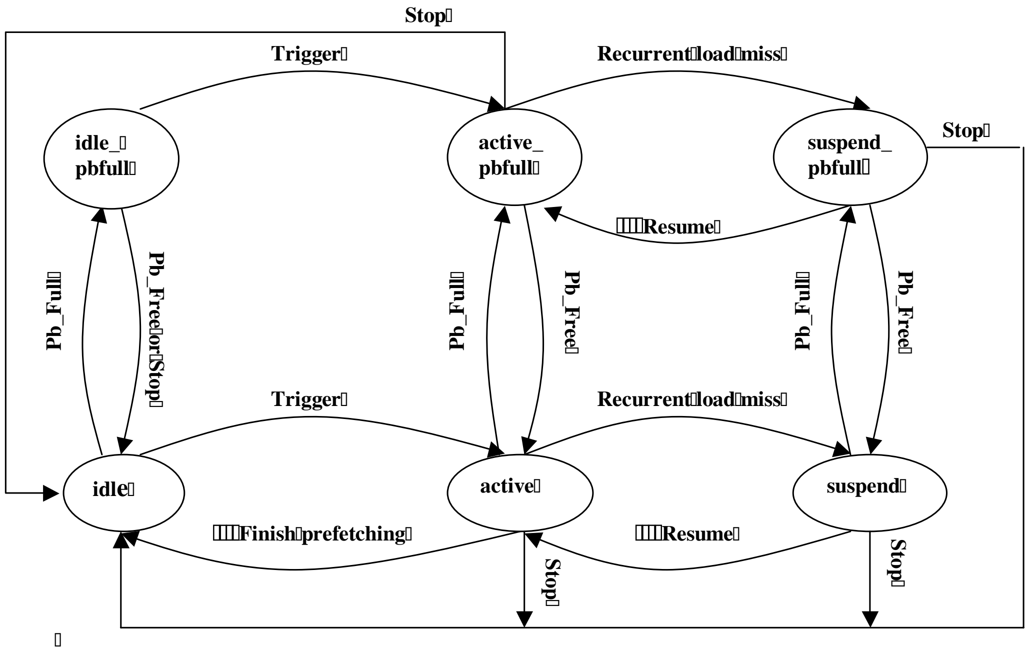


Figure 4.13: Complete PFE State Diagram

resumes execution when it observes a Resume message. When an engine in the active state observes a Pb_Full message, it suspends execution and moves to the active_pbfull state. A PFE that is in the idle and suspend state transfers to the idle_pbfull and suspend_pbfull state when it sees the Pb_Full message. This is to prevent a PFE from starting execution when it sees a Resume or Trigger message while the prefetch buffer is full. For example, an engine at the suspend_pbfull state moves to the active_pbfull state after receiving a Resume message and does not start execution until the Pb_Free message is observed. A PFE returns to the idle state when observing a Stop message.

I have described the mechanism used in the push architecture to avoid early and useless prefetches. Next, I present an architectural feature that reduces the effect of redundant prefetches.

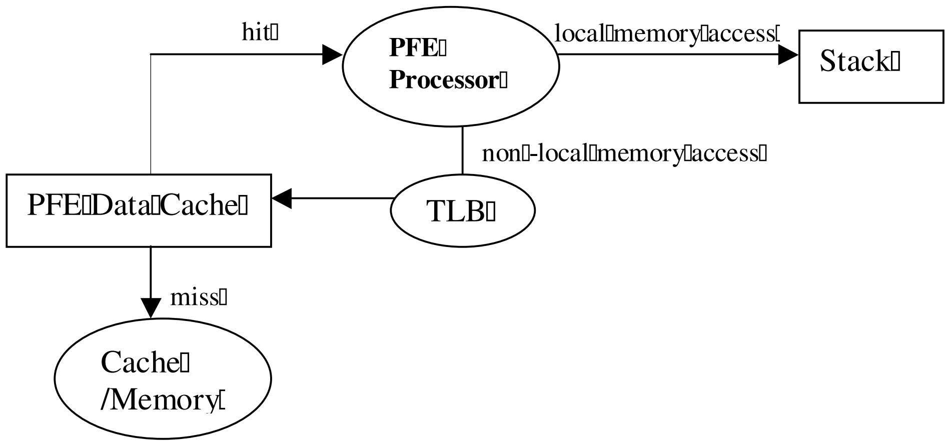


Figure 4.14: The PFE Block Diagram with a Data Cache

4.4 Reducing the Effect of Redundant Prefetches

One potential problem for the push scheme is that the cache blocks pushed up by the lower level engines could already exist in the upper level. These redundant prefetches could affect performance by wasting bus bandwidth or causing the PFE to run behind the main computation. Bandwidth is generally not the bottleneck for pointer-based applications because they are highly serial.

The potential performance impact of the PFE executing behind the main CPU is significant. Since memory accesses are satisfied more slowly in the lower levels of the memory hierarchy, if there are many redundant prefetches, the PFE may actually get behind the computation. 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. To capture this locality, I add a small fully-associative data cache which can be accessed in one cycle in the L2 and memory PFEs.

Figure 4.14 shows the modification of the PFE design after adding the data cache. A non-local memory request accesses the PFE data cache first. The PFE data cache is indexed using physical addresses. A load that hits in the local data cache is resolved in one cycle instead of the slower memory access latency. Prefetch requests that miss in the PFE data cache are then forwarded to the cache/memory controller, and the PFE data cache is updated when requested cache blocks are sent back to the engine.

The cache blocks found in the PFE data cache are also pushed up the memory hierarchy to ensure that the throttle mechanism described in Section 4.3 works properly. A prefetch that hits in the PFE data cache is not necessarily a redundant prefetch. If a cache block accessed by such a prefetch is not pushed up, a recurrent load that accesses this block will miss in the L1 and prefetch buffer. If the PFEs are running too far ahead of the CPU, the prefetch buffer could be full when this miss occurs. So this miss is treated as a hint that the PFEs are performing useless prefetches. As a result, the PFEs stop the current prefetching even though they are actually running ahead of the CPU. The downside of pushing all cache blocks up is wasting bus bandwidth. However, as mentioned before, bandwidth is generally not the bottleneck for the pointer-based application. In the next chapter, I use simulation results to verify this assertion.

So far I have presented four main design issues of the push architecture: the internal design of the PFE, the interaction among three PFEs, the throttle mechanism to synchronize the CPU/PFE execution and a technique to avoid redundant prefetches. Next, I describe the required modifications to the cache/memory controller to support the push model.

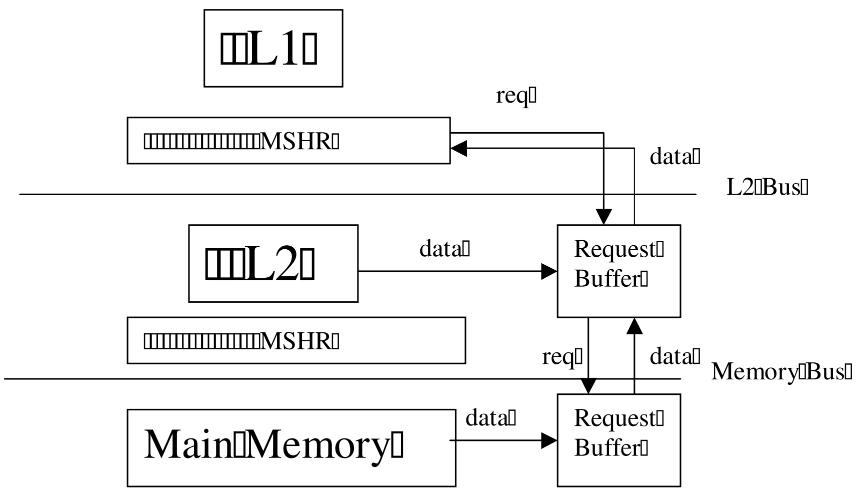


Figure 4.15: The Block Diagram of the Memory Hierarchy

4.5 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, I first describe the implementation details of the memory hierarchy. The block diagram of the memory hierarchy is shown in Figure 4.15. The L1 and L2 cache controllers include miss information/status holding registers (MSHR) for each miss that will be handled concurrently [36]. When a cache miss occurs, a MSHR is allocated to store the information 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 non-binding 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.

Chapter 5

Evaluation

In this chapter, I present simulation results that demonstrate the effectiveness of the proposed prefetching scheme. Section 5.1 describes the simulation framework. In section 5.2, I show the analysis of a microbenchmark to provide insight into the performance advantage of the push model over the conventional pull model. I then examine the prefetching scheme in detail using a set of pointer-based applications in section 5.3. I first present performance improvement achieved by the push architecture using a single-issue processor as the PFE assuming a perfect TLB. I then analyze how different PFE microarchitectures, the number of PFEs and the memory PFE clock rate affect performance. After that, I examine the impact of address translation. I finish by evaluating the push model performance for future processors.

5.1 Methodology

I evaluate the push model performance using cycle-by-cycle execution-driven simulation. I modified the SimpleScalar simulator [7] to include the PFE implementation. SimpleScalar models a dynamically scheduled processor using a Register Update Unit (RUU) and a Load/Store Queue (LSQ) [63]. 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) Write-back: supply results to dependent instructions, and 5) Commit: commit results

Memory Access Time	48 cycles
L2 Access Time	12 cycles
L2 Bus Speed / Width	400MHz / 128 bits
Mem Bus Speed / Width	400MHz / 64 bits

Table 5.1: Memory System Configuration

to the register file in program order, free RUU and LSQ entries.

The baseline processor 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 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 shared second level cache. Both 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. I derived the memory system parameters based on the Alpha 21264 design [33], which has an 800MHz CPU and 60ns DRAM access time. The memory system configuration is summarized in Table 5.1. I do not model interleaved memory. Pointer-based applications are highly serial; that is, very few memory requests can proceed in parallel. So the performance impact from using interleaved memory should not be significant.

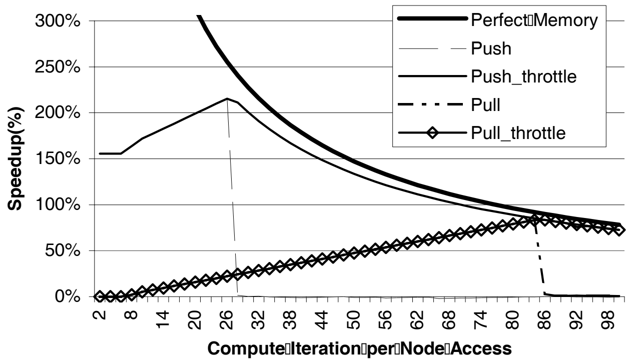
I first evaluate the performance of the push model on the 3_PFE architecture (i.e. attaching the PFE to each level of the memory hierarchy). The PFE is an in-order, single-issue processor, which runs at the same clock rate as the CPU. I believe that this is a reasonable configuration since an embedded processor like StrongARM-2 [39] is able to achieve 800MHz clock rate. I then vary the PFE issue width, speed and the number of PFEs used in the push architecture. I 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 [48] 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 I do not add additional cache/memory ports to support the push model, I assume that the cache and memory access latency remains the same as the base model. TLBs at different levels of the memory hierarchy are updated independently. I implement hardware managed TLBs with a 30 cycle miss penalty.

I 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. I use CProf [38] to identify kernels that contribute the most cache misses, and construct only these kernels by hand. As mentioned in Section 3.1, 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.

5.2 Microbenchmark Results

I begin by examining the performance of both prefetching models (push and pull) using a microbenchmark. This microbenchmark simply traverses a linked list of 32,768 32-byte nodes 10 times, see Figure 5.1(a). Therefore, all LDS loads are recurrent loads. I also include a small computation loop between each node access.

```
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

Figure 5.1: Microbenchmark

This allows us to evaluate the effects of various ratios between computation and memory access. To eliminate branch effects, I use a perfect branch predictor for these microbenchmark simulations.

Figure 5.1(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, I test 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 can examine the results shown in Figure 5.1(b) in three distinct phases. 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, both the push and pull models show larger reductions in memory stall time. When the number of compute iterations increases to 26, the push model reaches its peak performance, close to a 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 a 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 could have significant performance advantage over the pull model for applications with little computation between node accesses for the memory configuration tested. The push model may perform equally 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.

5.3 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 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, I use the Olden pointer-intensive benchmark suite [53] and Rayshade [35]. I compile the programs for the MIPS-I architecture using the GNU GCC compiler with opti-

Benchmark	Linked Data Structure	Input Parameter	Data Set Size	Traversal Kernel Size	L1 /L2 read miss rate
bh	octree	4K bodies	720KB	108B	3% / 45%
bisort	binary tree	250,000 numbers	1.5MB	24B	6% / 36%
em3d	list	2000 nodes, arity 10	1.6MB	92B	57% / 19%
health	list	5 levels, 500 iterations	925KB	32B	26% / 53%
mst	array of lists	1024 nodes	20KB	28B	16% /58%
perimeter	quadtree	4kx4k image	6.4MB	184B	9% / 70%
treeadd	binary tree	20 levels	32MB	68B	12/% /50%
tsp	list	100,000 cities	5.1MB	48B	5% / 48%
voronoi	binary tree	60,000 points	11MB	52B	2% / 15%
rayshade	list	Teapot data set	671KB	68B	13% / 88%
power	tree & list	10,000 nodes	313KB	-	0.7%/51%

Table 5.2: Benchmark Characteristics

mization flags -O2.

5.3.1 Benchmark Characterization

The Olden benchmark suit has been used in the past for studying the memory behavior of pointer-intensive applications [54, 56, 42]. Olden contains ten pointer-based applications. Health models the Columbia Health Care system. Bh implements the Barnes-Hut's hierarchical N-body algorithm. Bisort implements the Bitonic sorting algorithm. Em3d simulates electro-magnetic fields and processes lists of interacting nodes in a bipartite graph. Mst finds the minimum spanning tree of a graph. Perimeter computes perimeters of regions in images. Treeadd sums the values distributed on a binary tree. Tsp solves the traveling salesman problem. Voronoi computes the voronoi diagram of a set of points. Power solves the power system optimization problem. Rayshade is a real-world graphics application that implements a raytracing algorithm.

Table 5.2 summaries several important attributes of the benchmarks, includ-

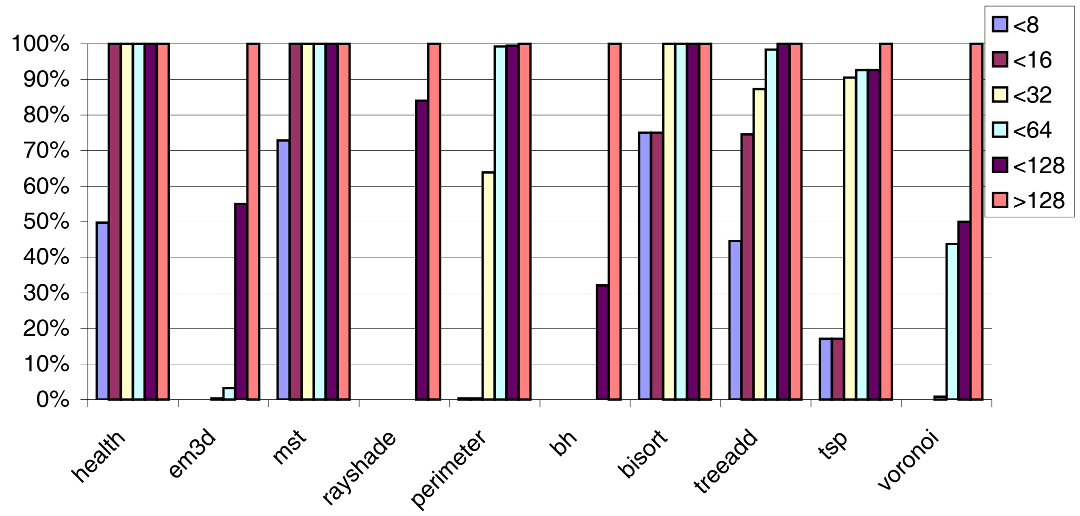


Figure 5.2: Cumulative Distance Distribution between Recurrent Loads

ing the types of linked-data structures, the L1/L2 cache miss ratio[1], the data set size and traversal kernel size used in this study. Note that I only list the data structure types for the traversal kernels executed by the PFEs. Appendix A lists the traversal kernels of this set of benchmarks. Since the power application has only 1% miss ratio, I do not evaluate this program further.

Another important program attribute that affects prefetching performance is the amount of computation available for overlapping with memory accesses. I quantify this parameter by measuring the number of instructions issued between recurrent loads – recurrent load distance. Figure 5.2 presents cumulative distribution of these distances. Em3d, rayshade and bh represent programs that have long computation available for overlapping with prefetches. 70% of recurrent loads for bh have more than 128 instructions issued in between. Health and mst represent programs with tight traversal loops. Less than 16 instructions are issued between recurrent loads for these two benchmarks.

[1]Table 5.2 lists the L2 local miss rate

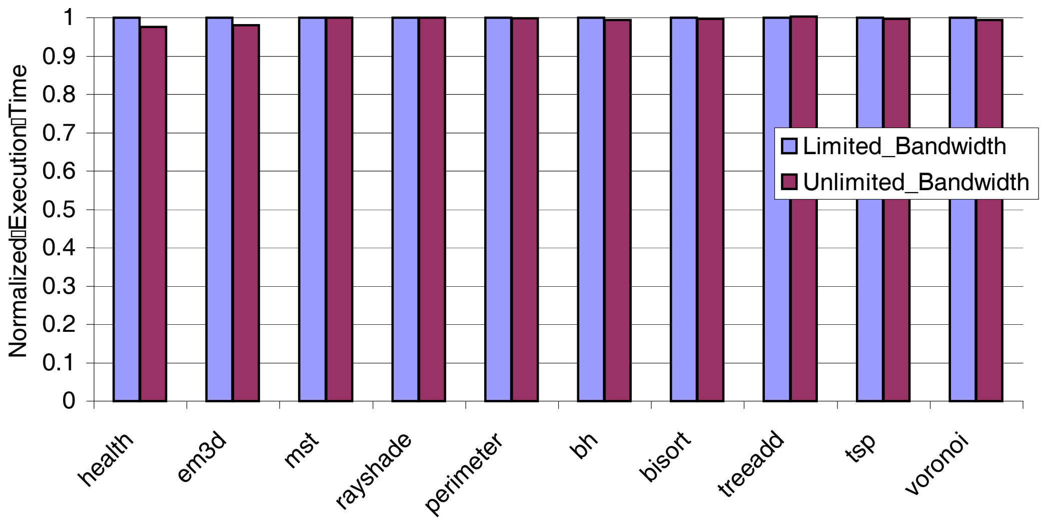


Figure 5.3: Bandwidth Evaluation

Recall that the mechanism used to reduce the performance impact of redundant prefetches (see Section 4.4) assumes that bandwidth is not the performance bottleneck for pointer-based applications. To verify this assertion, I measure performance of two configurations: one models bus contention (limited_bandwidth), and the other assumes that the L2/memory bus is always available (unlimited_bandwidth). Figure 5.3 shows execution time normalized to limited_bus. From the result we can conclude that bandwidth is not the performance bottleneck of this set of applications because limited_bandwidth achieves almost the same performance as unlimited_bandwidth.

In the next section, I evaluate the performance advantage of the push model over the traditional pull model for this set of benchmarks. Since the push model is designed to improve data cache performance, I first present simulation results assuming a perfect TLB. I then examine the TLB effect using different TLB configurations. Note that unlike the microbenchmark results, the branch prediction effect is considered.

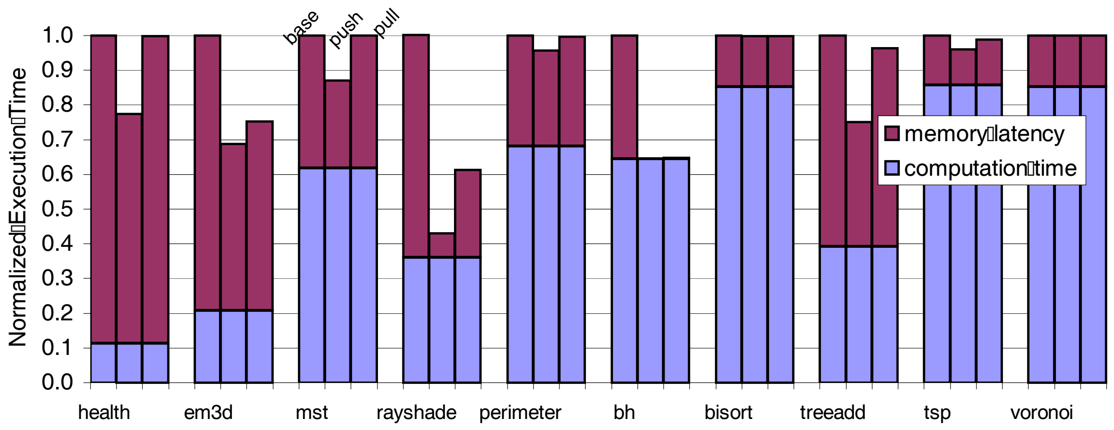


Figure 5.4: Performance Comparison between the Push and Pull Model

5.3.2 Performance Comparison Between The Push and Pull Model

Figure 5.4 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. I obtain the computation time by assuming a perfect memory system.

For benchmarks with tight traversal loops (health, mst), the push model is able to reduce between 26% and 34% of memory stall time (13% to 23% overall execution time reduction) while the pull model is not able to achieve any performance improvement as we expect. Treeadd also implements a tight traversal loop but has slightly longer recurrent distance compared to health and mst as shown in Figure 5.2. So we see that the pull model is able to achieve some speedup, but the push model still outperforms the pull model (25% vs. 4% execution time reduction). Perimeter traverses down a quad-tree in depth-first order, but has an unpredictable access pattern once it reaches a leaf. Therefore, I only prefetch the main traversal kernel. Although perimeter performs some computation down the leaves, it has very little computation to overlap with memory accesses when traversing the internal nodes. So the pull model is not able to achieve any speedup, but the push model reduces 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 for both the push and pull model. From Figure 5.4 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, and reduces execution time by 39% for rayshade. For em3d, the pull model achieves significant speed like bh and rayshade (31% reduction in memory stall time and 25% in execution time), but the push model only show slight performance improvement over the pull model (31% reduction in execution time).

When the pull model performs comparably to the push model, we expect performance close to a perfect memory system (see Section 5.2). 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 3.1, the prediction accuracy is low for this type of application. For tsp, we are able to identify several 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 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 models can reduce execution time by 3%. Voronoi uses pointers, but array and scalar loads cause most cache misses. Neither of the push and pull model is able to achieve any speedup because they do not prefetch those data types. Since bisort and voronoi do not benefit from the push model, I omit these applications in the following discussion.

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 a perfect memory system while the pull model is not able to deliver comparable performance.

5.3.3 Prefetch Coverage for the Push Model

To understand more about the prefetching behavior of the push model, I present prefetch coverage in Figure 5.5. Prefetch coverage measures the percentage of cache misses serviced by the prefetching mechanism. Fully hidden misses are

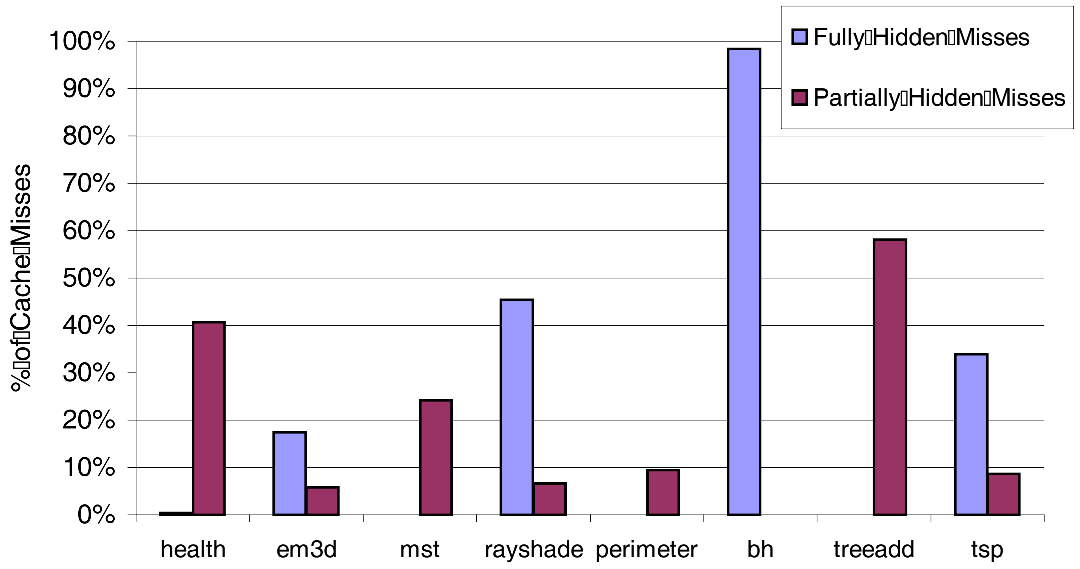


Figure 5.5: Prefetch Coverage for the Push Model

would-be cache misses that are eliminated because of prefetches. Partially hidden misses are those cache misses that merge with prefetch requests somewhere in the memory hierarchy. The latency for these misses is only partially hidden. For applications that have tight traversal loops, such as health, mst, treeadd and perimeter, all the performance gain from prefetching comes from partially hidden misses. Em3d, rayshade and bh have more computation between node accesses, so we see more total hidden misses. 98% of misses are completely removed for bh.

5.3.4 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, I evaluate the effect of the PFE data cache and throttle mechanism separately in Figure 5.6.

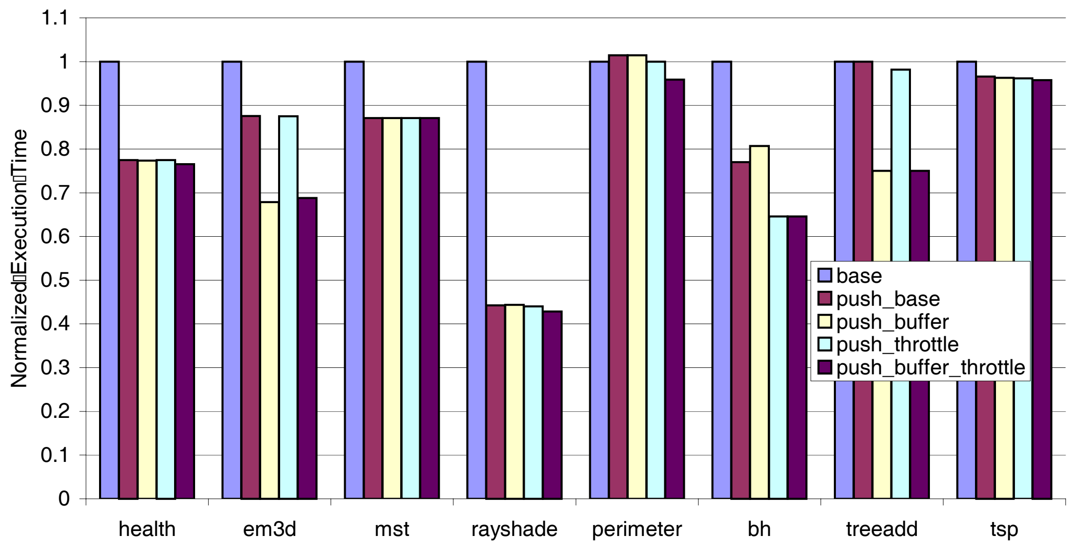


Figure 5.6: Effect of the PFE Data Cache and Throttle Mechanism

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 is not exclusive. The data caches can speed up PFE execution, while throttling slows down the PFE execution when they run too far ahead. Therefore, push_buffer_throttle shows the combined effect of both features, which are the previously presented results in Figure 5.4.

Effect of the PFE Data Cache

From Figure 5.6 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

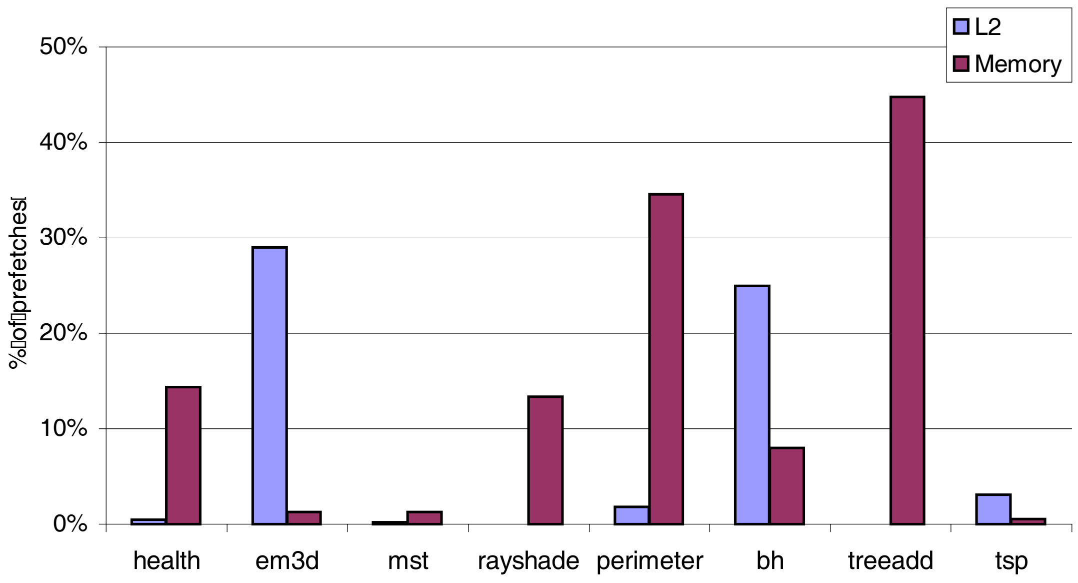


Figure 5.7: Redundant Prefetches from the L2/Memory Levels. The y-axis shows the percentage of prefetches that are redundant. This number is obtained by dividing the number of redundant prefetches by the total number of prefetches

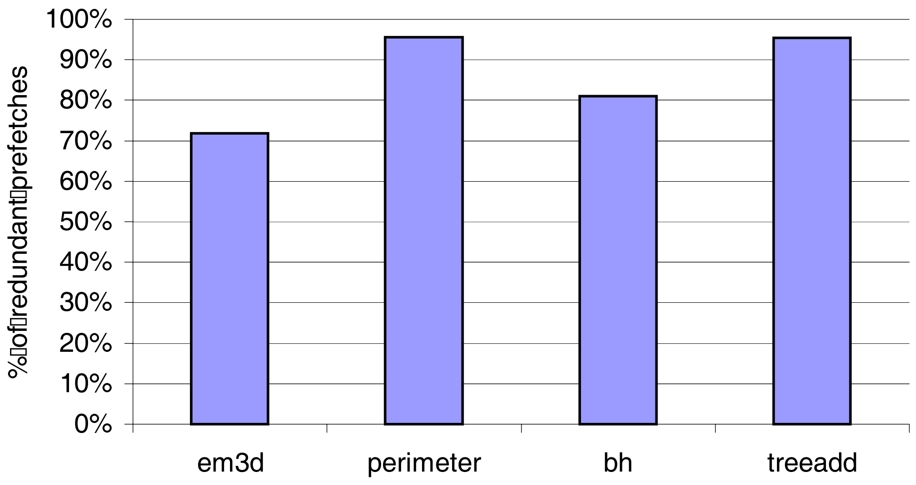


Figure 5.8: Effect of the PFE Data Cache. The y-axis shows the percentage of redundant prefetches that are PFE data cache hits

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 5.7. Bh also has a significant amount of redundant prefetches (33%). Treeadd, bh and perimeter are tree traversals in depth-first order, which can cause redundant prefetches as mentioned in Section 4.4. 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.

To evaluate how well the PFE data cache captures these redundant prefetches, I measure the percentage of redundant prefetches that are PFE data cache hits for these four benchmarks. The result is shown in Figure 5.7. For tree traversal applications (perimeter, bh and treeadd), the PFE data cache is able to capture between 80% to 100% of redundant prefetches. Em3d's access pattern is not as regular as traversing a tree in depth-first order, but the PFE data cache is still able to filter out 70% of 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).

Effect of the Throttle Mechanism

Figure 5.6 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.

I have demonstrated the performance advantage of the proposed push model over the conventional pull model. I have also shown that the two architectural features, the PFE data cache and throttle mechanism, successfully remove redundant and early prefetches. Next, I examine the performance impact of different PFE architectures.

5.3.5 Performance Impact of Different PFE Architectures

Simulation results presented so far use an in-order, single-issue processor as the PFE. This section examines how different PFE architectures affect performance. I first compare the performance of the specialized PFE with that of the programmable PFE examined above. I then look at the benefit of increasing the issue width of the programmable PFE.

Programmable vs. Specialized PFE

The specialized PFE design described in Section 4.1.1 is a highly optimized implementation for linked-list traversals. It executes traversal kernels out of order and gives a recurrent load priority if there are multiple loads ready at the same

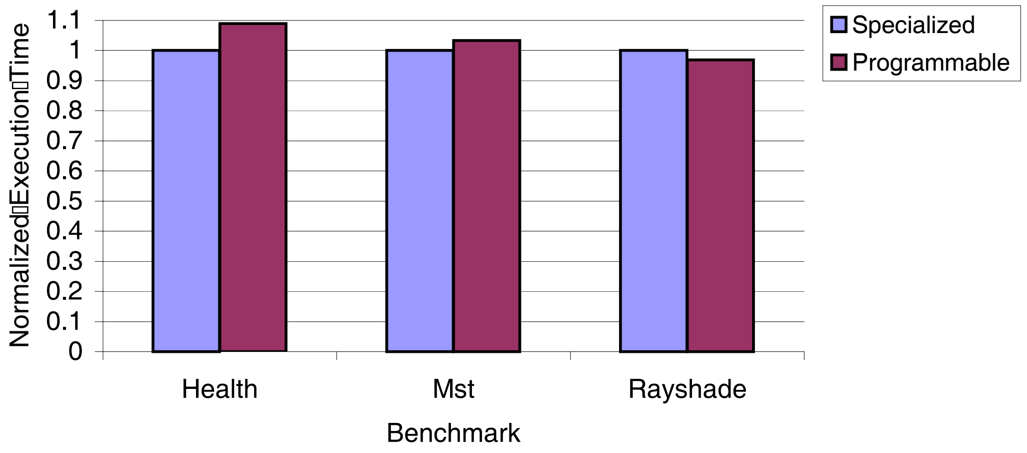


Figure 5.9: Programmable vs. Specialized PFE

cycle. The programmable PFE examined above has a flexibility advantage over the specialized hardware, but may not be able to deliver the same performance. In this section, I evaluate the possible performance loss from using the programmable PFE for three list-based applications. Note that although these applications do use tree structures, most cache misses come from traversing linked lists.

Figure 5.9 shows execution time normalized to that of the specialized PFE for these two designs. Simulation results show that the programmable PFE is able to deliver performance within 10% of the specialized hardware. For rayshade, it even outperforms the specialized PFE. Rayshade always traverses the whole linked list, but the fields of a LDS structure accessed cannot be determined statically. The programmable PFE is able to perform value comparisons to determine the traversal pattern at the run time, but the specialized PFE is not able to handle such dynamic behavior as well as the programmable one.

From this set of experiments we know that the programmable PFE only performs slightly worse than the specialized PFE, which is capable of traversing linked lists only. So using a general purpose processor core as the PFE is clearly

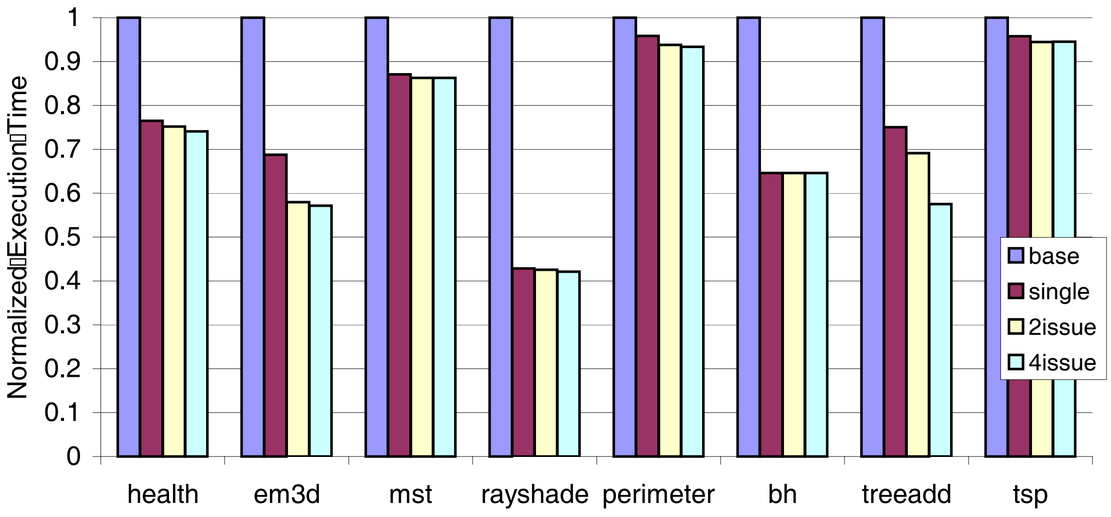


Figure 5.10: Effect of Wider Issue PFEs

a better design choice because it has a flexibility advantage over the specialized hardware.

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. This section evaluates if the push model can obtain further performance improvement using wider issue PFEs. Figure 5.10 shows 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 execution time by 11%. Treeadd 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

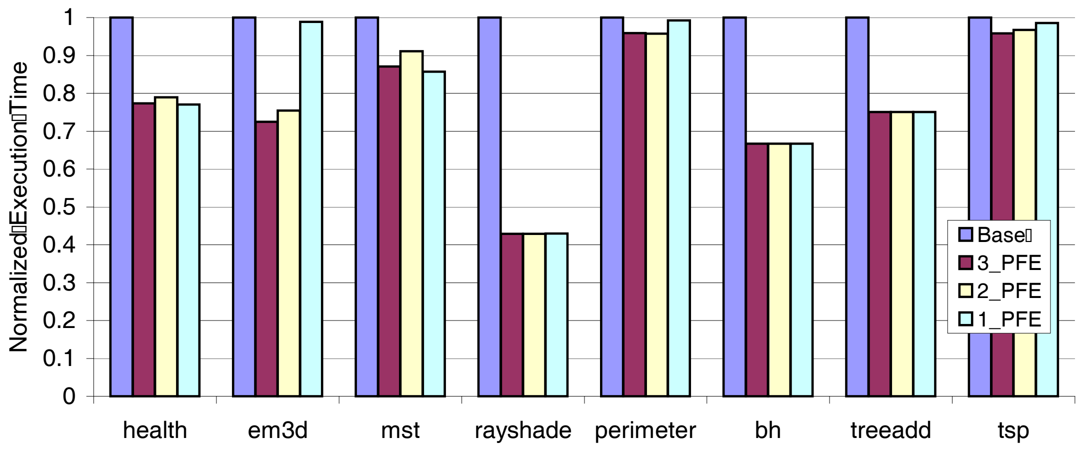


Figure 5.11: Variations of the Push Architecture

execution time by 9% more compared to a single-issue processor. Execution time is further reduced by 11% after increasing the issue width from two to four. Note that I 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.

5.3.6 Number of PFEs

This section evaluates the performance of various push architectures discussed in Section 3.3: 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 5.11 shows execution time of these three architectures normalized to the base configuration.

From Figure 5.11 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 5.3.2, 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 5.4.

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 3.3, 2_PFE only needs one PFE if the CPU is a multithreading processor, which is the trend for future processors [25]. Therefore, 2_PFE is the best design choice considering both cost and performance.

5.3.7 Speed of PFEs

In the above section, I have shown that 2_PFE is the best design choice. In this section, I evaluate if we can further reduce the hardware cost of 2_PFE by using a slower memory PFE. Simulations presented so far assume that the PFEs run

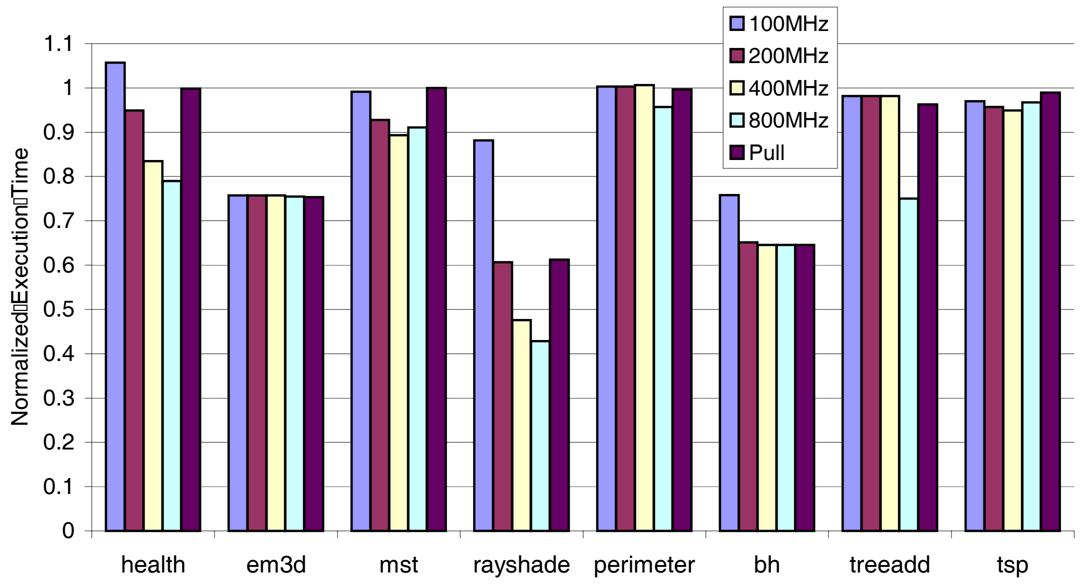


Figure 5.12: Effect of the PFE Clock Rate

at the same speed as the CPU (800MHz).

Figure 5.12 shows the execution time of 2_PFE normalized to the base model for four different memory PFE clock rates: 100MHz, 200MHz, 400MHz and 800MHz. I also list the pull model performance for comparison. For applications that the PFE data cache has a significant impact on performance (see Section 5.3.4), like treeadd and perimeter, the memory PFE needs to run at least the same clock rate as the CPU (800 MHz) to achieve better performance than the pull model. That is because a significant amount of loads are satisfied in one cycle instead of the long DRAM access latency. Em3d also benefits from the PFE data cache, but its performance is not affected by the memory PFE speed since it issues most of prefetches from the L1 level. For other benchmarks, a 400MHz memory PFE can perform comparably to a 800MHz PFE. 2_PFE with a 100MHz memory PFE is not able to outperform the pull model for most applications tested.

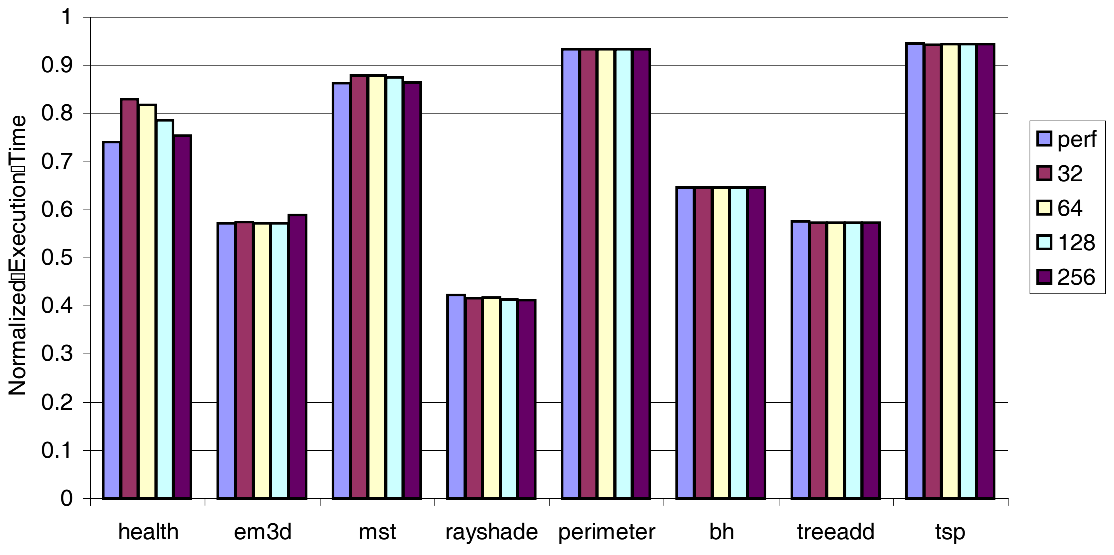


Figure 5.13: Impact of Address Translation

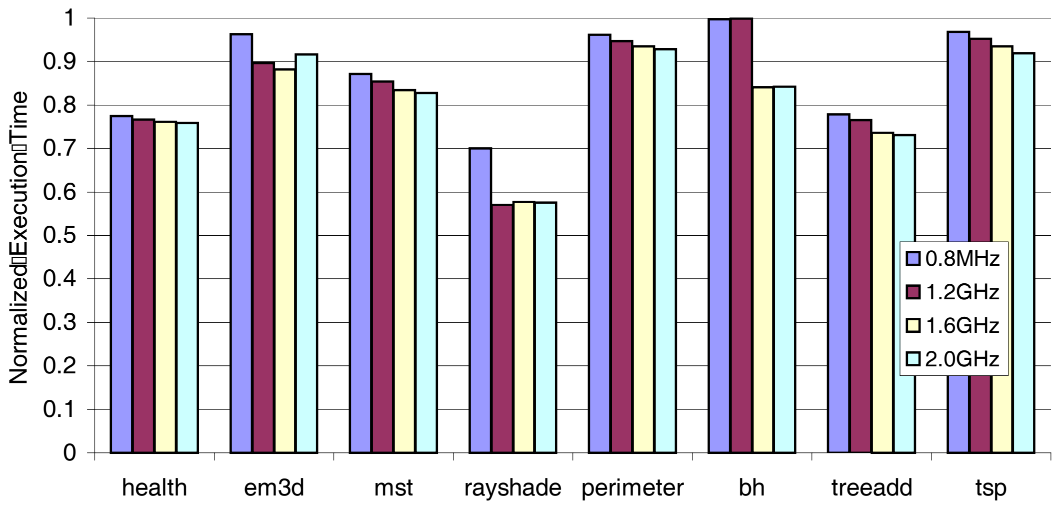
5.3.8 Impact of Address Translation

The proposed push architecture places a TLB with each prefetch engine. However, the results presented thus far assume a perfect TLB. This section evaluates the performance impact of TLB misses. Figure 5.13 shows the normalized execution time of the 3_PFE architecture using a 4-issue PFE for various TLB sizes (32, 64, 128 and 256). Health is the only benchmark significantly affected by TLB performance. Using a 256-entry TLB, health achieves the same performance as the perfect TLB. However, even with a 32-entry TLB (17% TLB miss ratio), the proposed prefetching scheme is still able to reduce execution time by 17%. The push model could also prefetch TLB entries, but remains future work.

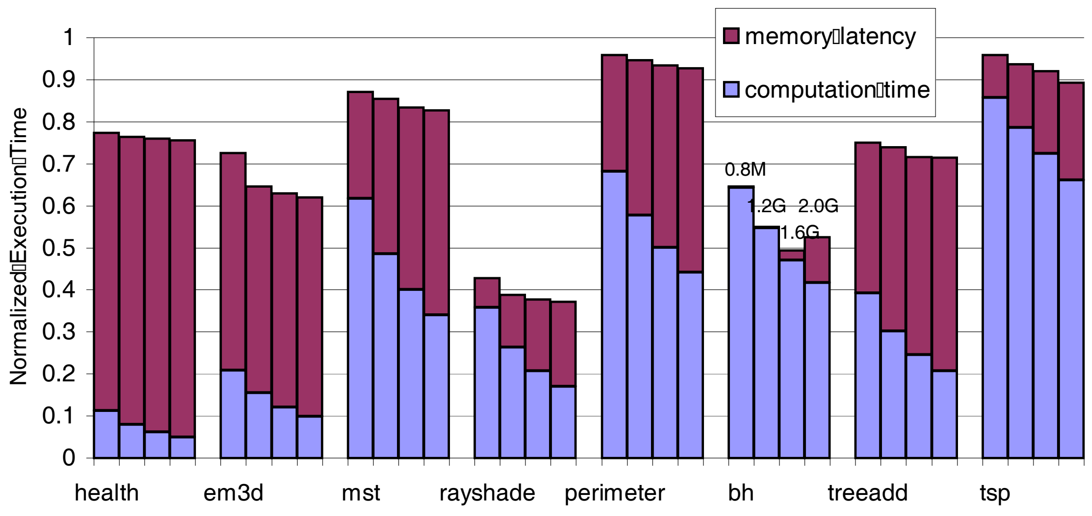
5.3.9 Performance Prediction of the Push Architecture for Future Processors

In this section, I examine the performance advantage of the push model over the traditional pull model as the performance gap between the CPU and memory system continues to grow using the 3_PFE architecture. I perform two sets of experiments. The first set only increases the CPU clock rate. In the second set of experiments, the memory system scales with the CPU speed except for the DRAM access time. Recall that the ratio of the DRAM access time to round-trip memory latency determines the relative performance of the push and pull model (see Section 3.2). The larger the ratio is, the smaller performance improvement the push model can achieve over the pull model. So the second set of experiments provides the lower bound of the push model performance.

Figure 5.14 shows the results of the first set of experiments where only the CPU gets faster. I first evaluate the relative performance of the push and pull model. Figure 5.14(a) shows execution time of the push model normalized to the pull model. Since the ratio of the DRAM access time to round-trip memory latency remains the same across different CPU configurations, the relative performance of the push and pull model does not have significant variation as expected except for em3d, bh and rayshade. The push model shows a larger performance improvement over the pull model as the CPU clock rate increases for these three benchmarks (from 0.8MHz to 1.2GHz for em3d and rayshade; from 1.2GHz to 1.6GHz for bh). Recall that these three benchmarks have longer computation between node accesses, so the pull model is able to achieve more speedup compared to other benchmarks (see Figure 5.4). As memory latency continues to grow, the amount of computation needed to hide memory latency is larger, and the pull model becomes less effective. Although it has impact on both the push and pull



a) Relative Performance of the Push and Pull Model. The y-axis shows execution time of the push model normalized to the pull model.



b) Performance of the Push Model. The y-axis shows execution time of the push model normalized to the base model.

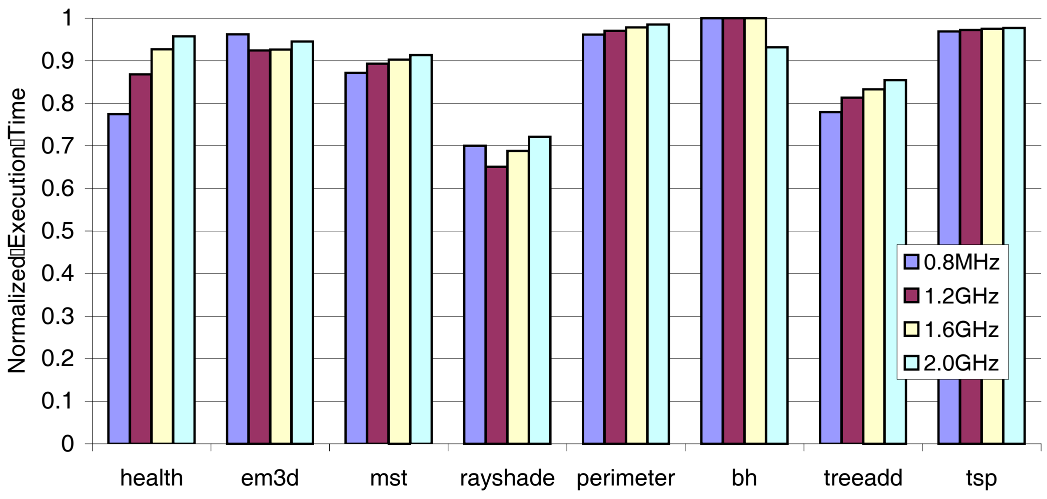
Figure 5.14: Performance Trend for Future Processors (1). Only the CPU clock rates increase.

model, it affects the push model at a slower rate, and thus the relative benefit of the push model increases.

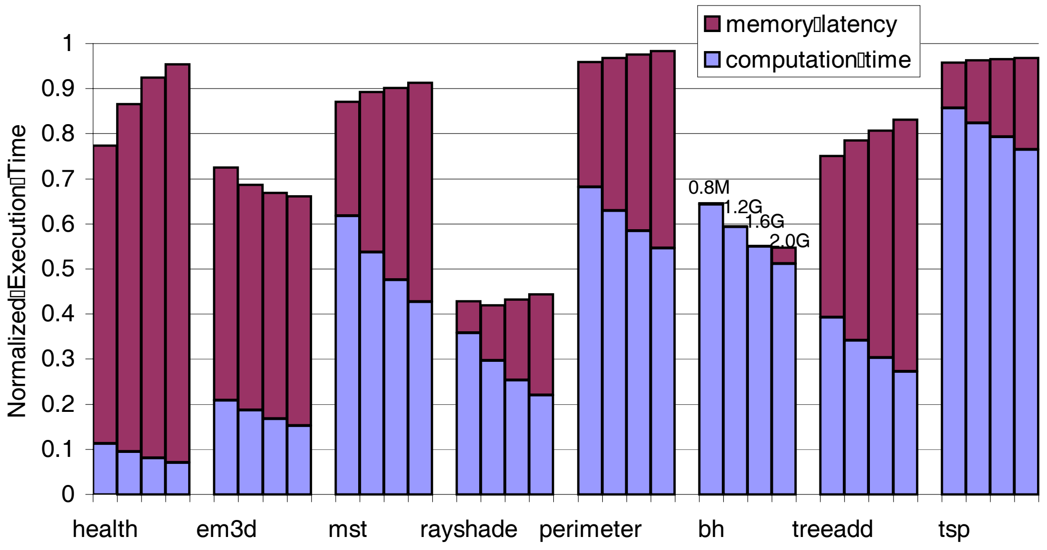
I examine how the push model affects overall performance for future processors in Figure 5.14(b). The y-axis shows execution time of the push model normalized to the base model. Execution time is divided into two components, computation time and memory stall time. We can see that as the CPU gets faster, the push model is less effective in reducing memory stall time (longer computation is needed to hide memory latency), but overall performance improvement increases because memory stall time becomes a larger portion of overall execution time (Amadahl's law). For bh, the push model reduces 100% of memory stall time and 36% of overall execution time for a 0.8MHz CPU. When the CPU clock rate increases to 2.0GHz, the push model only reduces 81% of memory stall time, but reduces execution time by 47%.

Figure 5.15 shows the results of the second set of experiments where the memory system scales with the CPU speed except for the DRAM access time. The relative performance of the push and pull model is shown in Figure 5.15(a). The y-axis shows execution time of the push model normalized to the pull model. We can see that normalized execution time of the push model increases as the CPU gets faster because the DRAM access time contributes more to the round-trip memory latency. However, the push model shows a decrease in normalized execution time as the CPU clock rate increases from 0.8MHz to 1.2GHz for em3d and rayshade, and from 1.6 to 2.0GHz for bh. As mentioned above, the pull model becomes less effective as memory latency becomes larger. Since these results show the lower bound of the push model performance, we know that the push model will retain its performance advantage over the pull model for future processors.

The overall performance impact of the push model for this set of experiments



a) Relative Performance of the Push and Pull Model. The y-axis shows execution time of the push model normalized to the pull model.



b) Performance of the Push Model. The y-axis shows execution time of the push model normalized to the base model.

Figure 5.15: Performance Trend for Future Processors (2). The memory system scales with the CPU clock rate except for the DRAM access time

is shown in Figure 5.15(b). The y-axis shows execution time of the push model normalized to the base model. For most benchmarks we see a slight decrease of performance improvement from the push model as the CPU gets faster because the push model loses some of its performance advantage as explained above. But the push model can still provide significant speedup for many benchmarks even when the CPU clock rate increases to 2.0GHz (2% to 55% execution time reduction).

Chapter 6

Related Work

This chapter summaries research related to this thesis work. Early prefetching techniques are designed for regular applications in which address streams have arithmetic regularity. I first review these studies. It is hard to apply this type of prefetching techniques to irregular applications because address regularity no longer exists. Recently, there have been several studies exploiting other program attributes to improve cache performance of irregular applications, including prefetching and data/computation reorganization. I discuss them in Section 6.2. The push architecture proposed in this thesis uses a separate processor to execute traversal kernels ahead of the CPU. In Section 6.3, I summarize studies that use decoupled architecture to improve memory system performance. The last category of research that is related to this thesis is combining processing power and memory in the same chip. These studies are discussed in Section 6.4.

6.1 Prefetching for Regular Applications

Early data prefetching research, both in hardware and software, focuses on applications that use array data structure and have regular access patterns. On the hardware side, Jouppi [30] proposes to use stream buffers to prefetch sequential streams of cache lines based on cache misses. The original stream buffer design can only detect unit-stride. Palacharla et al. [47] extend this mechanism to detect non-unit stride off-chip. Several hardware prefetching schemes [5, 11, 21, 59, 20] make use of the program counter of the load/store instructions. Baer et al. [5, 11]

propose to use a reference prediction table (RPT) to detect the stride associated with individual load/store instructions. The RPT is searched using the look-ahead program counter that runs ahead of the regular PC. A prefetch request is issued once a matching entry is found. The Tango system [49] extends Baer's scheme to issue prefetches more effectively for superscalar processors. Tango generates more than one prefetch request in one cycle to supply data fast enough for multiple-issue processors. It also implements a prefetch request controller for filtering out redundant prefetches.

Software prefetching [8, 51, 34, 45, 44, 12, 24] exploits compile-time information to insert prefetch instructions in a program. Porterfield et al. [51, 8] propose an algorithm to automatic insert prefetch instructions into loops. Their schemes only prefetch data one iteration ahead. Klaiber et al. [34] extend Porterfield's algorithm to prefetch data more than one iteration ahead if necessary. They estimate the prefetch distance considering the amount of computation in a loop and memory latency. Mowry et al. [45] improve on the above software prefetching approaches by performing locality analysis to avoid unnecessary prefetches. Chen et al. [12] propose a data prefetching framework for superscalar processors. They use compiler to insert prefetch instructions for non-scientific codes and a prefetch buffer to avoid cache pollution.

Irregular applications present two challenges to any prefetching technique: address irregularity and the pointer-chasing problem. Next, I discuss prefetching techniques that try to overcome these challenges. Some researches seek to improve cache performance for irregular applications by reorganizing the computation sequence or data layouts. I also describe this type of techniques in the next section.

6.2 Mechanisms for Improving Memory Performance for Irregular Applications

In this section, I discuss two types of techniques for improving memory performance of irregular applications: prefetching and data/computation reorganization.

Prefetching

Correlation-based prefetching [29, 2, 10] uses the current state of the address stream to predict future references for applications that have irregular access patterns. Joseph et al. [29] build a Markov model to approximate the miss reference string using a transition frequency. This model is realized in hardware as a prediction table where each entry contains an index address and four prioritized predictions. Alexander et al. [2] also implement a table-based prediction scheme similar to [29]. However, it is implemented at the main memory level to prefetch data from the DRAM array to SRAM prefetch buffers on DRAM chips. Correlation-based prefetching can capture complex access patterns, but it has two main shortcomings. First, since it still relies on address history to predict future addresses, prediction accuracy decreases if applications have dynamically changing data structures or traversal orders. Second, the size of the prediction table needs to be proportional to the working set size so that it can capture sufficient address history to achieve satisfactory address prediction accuracy.

Another class of data prefetching mechanism focuses specifically on pointer-intensive applications. They explore data structure information to construct solutions. The Spaid scheme [40] proposed by Lipasti et al. is a compiler-based pointer prefetching mechanism. It inserts prefetch instructions for the objects pointed by pointer arguments at call sites. Zhang et al. [69] use object information to guide prefetching for irregular applications in shared-memory multiprocessors. Their method first binds data that are accessed close in time into groups and then prefetches the whole group once a memory line in a group is accessed.

Luk et al. [42] propose three compiler-based prefetching algorithms, greedy, history-pointer and data-linearization prefetching. Greedy method prefetches all child nodes once the parent node is obtained. History-pointer prefetching creates pointers that provide direct access to non-adjacent nodes based on recent traversals of the LDS. This approach is only effective when the LDS traversal order does not change frequently. This method incurs both space and execution overhead for installing history pointers. Data-linearization prefetching maps heap-allocated nodes into contiguous memory locations so future addresses can be correctly predicted without pointer dereferences. However, this method is only effective when the LDS traversal order is the same as the creation order.

Mehrotra et al. [43] extend stride detection schemes [5] to capture both linear and recurrent access patterns. Their approach can only prefetch recurrent loads. Roth et al. [54] propose a dependence based prefetching mechanism (DBP) that dynamically captures LDS traversal kernels and issues prefetches. In this scheme, they only prefetch a single instance ahead of a given load. The other constraint is that they perform wavefront tree prefetching (i.e. all child nodes are prefetched once a node is obtained) regardless of the traversal order. This is a pull-based prefetching mechanism, so its performance is limited by the conventional pull-based data movement.

To overcome the pointer-chasing problem, Roth et al. [56] propose a jump-pointer prefetching framework similar to history-based prefetching proposed by Luk et al. [42]. They provide three methods to install jump pointers: software-only, hardware-only and a cooperative software/hardware technique. Even though their hardware mechanism incurs lower overhead for installing jump pointers compared to history-based prefetching, its performance is still limited if the LDS structure or traversal order changes frequently. The other constraint of their approach is that the prefetch distance is fixed for all applications. Karlsson et al. [32] present a prefetch array (PA) approach, which aggressively prefetches all possible nodes a few iterations ahead. The PA mechanism outperforms jump pointer prefetching when the LDS traversal order is not known a priori.

Data and Computation Reorganization

Some studies seek to improve performance of pointer-based applications by reordering the computation sequence or changing data layout. Chilimbi et al. [14] present automatic methods for two cache-conscious data placement techniques, clustering and coloring. Clustering packs data structure elements likely to be accessed successively into a cache block to improve spatial and temporal locality. Coloring maps contemporaneously accessed elements to non-conflicting regions of the cache to reduce conflict misses. The reorganization process happens at run time. For applications that have objects larger than the block size, Chilimbi et al. [13] propose two automatic methods to reorder the internal layout of a structure: structure splitting and field reordering. Ding et al. [18] also propose two run-time computation and data transformations. Locality grouping reorders computation to improve program temporal locality. The second technique, dynamic data packing, reorganizes data to improve program spatial locality.

Reordering techniques prove to be effective in improving program locality. However, for applications that change structures dynamically, reorganization overhead could be significant. The other shortcoming of these techniques is that they cannot eliminate capacity misses.

6.3 Decoupled Architecture

The third type of studies related to this thesis is decoupled architecture that uses a separate processor for memory accesses. Structured memory access [50] and the Decoupled Access Execute [62] separate execution into a memory access and a computation process that run on different processors or functional units. Both techniques try to overlap demand memory requests with computation. VanderWiel et al. [66] and Brent [6] propose a separate processor for prefetching purpose. The prefetching mechanism suggested by VanderWiel et al. [66] targets at regular applications; Brent's approach aims at improving the instruction cache performance.

Recently, several studies [4, 15, 19, 58, 64, 70, 41] 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. Roth et al. [58] and Zilles et al. [70] use program traces to identify speculative slices and execute them as helper threads in a simultaneous multithreading processor (SMT) [65]. The former implements register integration technique [57] to allow the main thread to use results produced by helper threads. The latter provides several optimizations to reduce the size of speculative slices. The pre-execution framework proposed by Collin et al. [15] differs from the above two approaches in that it allows a helper thread to spawn another speculative thread. Luk [41] identifies speculative slices in the program source code level and pre-executes multiple data streams simultaneously. Annavaram et al. [4] suggest constructing speculative slices at run time and use a separate processor to execute them. Slipstream Processors [64] executes two copies of one program (A-stream and R-stream) in a two-way chip multiprocessor. The length of A-stream is reduced by removing "ineffectual" instructions observed in R-stream execution. A-stream runs faster than R-stream because it is shorter. The results of A-stream are communicated to R-stream. These pre-execution approaches are essentially the pull-based prefetching that I evaluate. They should be able to obtain additional performance improvement by taking advantage of the push model proposed in this thesis work.

Crago et al. [16] propose a hierarchical decoupled architecture that adds a processor between each level of the memory hierarchy. This architecture employs three separate processors. Computation Processor performs the main computation of the program. Access Processor is responsible for address calculations and loading data from the L1 cache to registers. Cache Management Processor issues memory requests to the L1 cache, which is similar to the L1 PFE in the push architecture. So their framework is still based on the traditional pull model, and they only study array-based applications.

6.4 Processor-In-Memory Architecture

Because of the advance of a Merged Logic DRAM (MLD) process [52], combining processing power and memory in the same chip becomes a feasible approach to bridge the CPU-memory performance gap. The Berkley IRAM group [48] integrates the main processor with DRAM on a chip. The Active Page [46] and FlexRAM [31] integrate multiple computation elements in DRAM chips to exploit data parallelism in applications. This thesis differs from the above studies in that the PFEs do not perform computation except for address calculation and simple value comparison for control flow, and the target applications are pointer intensive applications, that are highly serial. Impulse [9] 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 thesis.

Concurrently with this thesis work, Hughes [28] evaluates memory-side prefetching in multiprocessor systems. His prefetching scheme is equivalent to the 1_PFE architecture studied in this thesis. His study 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.

Chapter 7

Conclusion

This thesis investigates the memory system performance of pointer-based applications. Prefetching is commonly used to hide memory latency by bringing data close to the CPU earlier than demand fetches. However, prefetching performance for pointer-based application is often limited because of the pointer-chasing problem. I observe that the impact of pointer-chasing on prefetching performance could be reduced if we can perform pointer-dereferences at the lower levels of the memory hierarchy and pro-actively push data up to the processor. In this thesis, I propose the push architecture to realize this novel data movement.

The push architecture is a cooperative hardware/software prefetching framework. 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.

There are four main design issues to implement this push architecture:

1. Microarchitecture of the PFE

   I present two PFE designs. One is the specialized PFE for linked-list traversal. The other is the programmable PFE design using a general purpose processor.

2. Interaction among PFEs

   I establish a general interaction protocol among three PFEs, particularly how a PFE suspends and resumes execution. I describe a mechanism to ensure that PFEs at the higher levels can resume execution properly for tree traversals.

3. Reducing the effect of redundant prefetches

   I use a small data cache in the L2 and memory PFE to filter out redundant prefetches. The PFE data cache captures recently accessed cache blocks by the PFE. Only prefetches that miss in the PFE data cache access the L2 cache and main memory.

4. The throttle mechanism

   I present an adaptive 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.

I perform execution-driven simulation to evaluate the push model performance. The simulation results show that the push architecture does have a significant performance advantage over traditional pull-based prefetching. For applications in which the pointer-chasing problem has most impact (i.e. applications with tight traversal loops), pull-based prefetching is not able to achieve any speedup, while the push model is able to reduce execution time by 13% to 23% (using a single, in-order issue PFE). For applications that have longer computation between node accesses, the push architecture can even perform comparably to a perfect memory system, while pull-based prefetching can only reduce memory stall time by 60%.

Simulations reveal that the proposed throttle mechanism successfully prevents the PFEs from running too far ahead of the CPU. Throttling increases the prefetch buffer hit ratio (the percentage of cache blocks accessed by the CPU) from 7% to nearly 100% for bh, which has the longest computation between node accesses among the benchmarks tested. The PFE data cache is proved to be very effective at filtering out redundant prefetches. Experiments show that 70% to 100% of redundant prefetches are captured in the PFE data cache.

I also evaluate how the PFE microarchitecture influences prefetching performance. Simulation results show that a simple, single-issue, in-order PFE is sufficient to provide significant speedups for most of benchmarks tested. For list-based applications, the programmable PFE is able to deliver performance within 10% of that of the specialized hardware. Increasing the issue width of the programmable PFE can further improve performance. I look at two ways to reduce the hardware cost of the push architecture: one is to employ fewer PFEs and the other is to use a slower memory PFE. Simulations 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. 2_PFE with a 400MHz memory PFE can achieve performance close to that of a 800MHz memory PFE for applications that the PFE data cache has little effect on (assuming a 800MHz CPU). 2_PFE with a 100MHz memory PFE is not able to outperform the pull model for most applications tested.

Simulation results also reveal that the push architecture continues to outperform the pull-based prefetching scheme for future processors. As memory latency continues to grow, the amount of computation needed to hide memory latency increases. Therefore, the pull model becomes less effective. Although longer memory latency impacts both the push and pull model, it affects the push model at a slower rate.

There are several extensions to the proposed prefetching framework that I would like to pursue in the future:

1. Automatic LDS traversal kernel generation

   I currently construct LDS traversal kernels manually. Compiler techniques can be used to construct traversal kernels automatically. Luk et al. [42] have already shown that type declaration and control structure information can be used to identify recurrent loads of LDS traversal successfully. By applying dependence analysis, a complete LDS traversal kernel can be constructed during compile time.

2. TLB prefetching

   The current push architecture is designed to improve the data cache performance only. After considering the TLB miss effect, overall performance improvement decreases. However, the push model can be easily extended to prefetch missing TLB entries as well. In the current implementation, TLBs at different levels of the memory hierarchy are updated independently. The push model can improve the TLB performance by pushing a missing TLB entry up the memory hierarchy when a miss occurs at the lower level TLBs.

3. The push model on multiprocessor systems

The current push architecture pushes data from lower levels of the memory hierarchy to the CPU for uniprocessor systems. The push model can be extended to prefetch data from remote memory modules for multiprocessor systems. A network processor can serve as a prefetch engine if it has the ability to execute LDS traversal kernels. The design challenge of employing the push model on multiprocessor systems is that the interaction among the PFEs is more complicated because a LDS could be distributed over several memory modules.

4. Applying the push model to other data structures

   In this thesis, I only study pointer-based applications. The proposed push architecture is also applicable to other data structures, such as the sparse matrix.

Appendix A

LDS Traversal Kernels of Olden Benchmark and Rayshade

Health

```
void check_patients_waiting
     (struct Village *village, struct List *list)
{
  int                        i;
  struct Patient             *p;
  struct List                *next;

  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(list);
  /* LDS TRAVERSAL */
  while (list != NULL)
    {
      next = list->forward;
      i = village->hosp.free_personnel;
      p = list->patient;
      if (i > 0)
      {
          village->hosp.free_personnel--;
          p->time_left = 3;
          p->time += p->time_left;
          removeList(&(village->hosp.waiting), p);
          addList(&(village->hosp.assess), p);
      }
      else
      {
          p->time++;
      }
      list = next;
    }
}
```

(a) Source Codes

```
void kernel(struct List *list)
{
  struct Patient           *p;
  struct List *next;

  while (list != NULL)
    {
      next = list->forward;
      p = list->patient;
      if (p)
      {
        data_load(p->time_left);
        data_load(p->time);
      }
      list = next;
    }
}
```

(b) Traversal Kernel

Mst

```
void *HashLookup(unsigned int key, Hash hash)
{
  int j;
  HashEntry ent;

  j = (hash->mapfunc)(key);
  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_reg_5(ent->key);
  write_root_reg(hash->array[j]);
  /* TRAVERSAL LOOP */
  for (ent = hash->array[j]; ent && ent->key!=key; ent=ent->next) ;
  if (ent) return ent->entry;
  return NULL;
}
```

(a) Source Codes

```
void kernel(HashEntry ent, int key)
{
  HashEntry next;

  while (ent)
  {
     next = ent->next;
     if (ent->key == key)
       break;
     ent = next;
  }
}
```

(b) Traversal Kernel

Barnes-Hut

```
void hackgrav(bodyptr p, nodeptr rt)
{
  hgstruct hg;

  hg.pskip = p;
  SETV(hg.pos0, Pos(p));
  hg.phi0 = 0.0;
  CLRV(hg.acc0);
  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(rt);
  /* LDS TRAVERSAL */
  hg = walksub(rt, hg);
  Phi(p) = hg.phi0;
  SETV(New_Acc(p), hg.acc0);
}

hgstruct walksub(nodeptr p, hgstruct hg)
{
  register int k;
  register nodeptr r;

  if (Type(p) != BODY) {
    for (k = 0; k < NSUB; k++) {
      r = Subp((cellptr) p)[k];
      if (r != NULL)
        hg = walksub(r, hg);
    }
  }
  else if (p != (nodeptr) hg.pskip)   {
    hg = gravsub(p, hg);
  }
  return hg;
}
```

a) Source Codes

```
void walksub_kernel(nodeptr p)
{
  register int k;
  register nodeptr r,r1;


  if (!p) return;

  r1 = Subp((cellptr) p)[0];

  if (Type(p) != BODY) {
    if (r1 != 0)
      walksub_kernel(r1);
  }
  if (Type(p) != BODY)
    {

      for (k = 1; k < NSUB; k++) {
        r = Subp((cellptr) p)[k];
        if (r != 0)
         walksub_kernel(r);
      }
    }
}
```

b) Traversal Kernel

Em3d

```
int main(int argc, char *argv[])
{
  int i;
  graph_t graph;

  dealwithargs(argc,argv);
  graph=initialize_graph();
  print_graph(graph);
  for (i = 0; i < iters; i++)
    {
      /* INITIALIZE PFE */
      write_kernel_id(1);
      write_root_reg(graph.e_nodes);
      /* LDS TRAVERSAL */
      compute_nodes(graph.e_nodes);
      /* INITIALIZE PFE */
      write_root_reg(graph.h_nodes);
      /* LDS TRAVERSAL */
      compute_nodes(graph.h_nodes);
      fprintf(stderr, "Completed a computation phase: %d\n", i);
      print_graph(graph);
    }
}

void compute_nodes(node_t *nodelist)
{
  int i;

  for (; nodelist; nodelist = nodelist->next)
  {
    for (i=0; i < nodelist->from_count; i++)
      {
        node_t *other_node = nodelist->from_nodes[i];
        double coeff = nodelist->coeffs[i];
        double value = other_node->value;
        nodelist->value -= coeff * value;
      }
    }
}
```

a) Source Codes

```
void kernel(node_t *nodelist)
{
  int i;

  for (; nodelist; nodelist = nodelist->next)
    {
      for (i=0; i < nodelist->from_count; i++)
        {
          node_t *other_node = nodelist->from_nodes[i];
          data_load (nodelist->coeffs[i]);
          date_load (other_node->value);
          data_load (nodelist->value);
        }
    }
}
```

b) Traversal Kernel

Perimeter

```
int main(int argc, char *argv[])
{
  QuadTree tree;
  int count;
  int level;

  level = dealwithargs(argc,argv);
  tree=MakeTree(4096,0,0,NULL,southeast,level);
  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(tree);
  /* LDS TRAVERSAL */
  count=CountTree(tree);
  /* INITIALIZE PFE */
  write_kernel_id(2);
  write_root_reg(tree);
  /* LDS TRAVERSAL */
  count=perimeter(tree,4096);

  chatting("perimeter is %d\n",count);
  exit(0);
}

int CountTree(QuadTree tree)
{
  QuadTree nw,ne,sw,se;

  nw = tree->nw; ne = tree->ne; sw = tree->sw; se = tree->se;
  if (nw==NULL && ne==NULL && sw==NULL && se==NULL)
    return 1;
  else
```

```
    return CountTree(nw) + CountTree(ne) +
           CountTree(se) + CountTree(sw);
}

int perimeter(QuadTree tree, int size)
{
  int retval = 0;
  QuadTree neighbor;

  if (tree->color==grey)
    {
      retval += perimeter(tree->nw, size/2);
      retval += perimeter(tree->ne, size/2);
      retval += perimeter(tree->sw, size/2);
      retval += perimeter(tree->se, size/2);
    }
  else if (tree->color==black)
    {
      /* South */
      neighbor=gtequal_adj_neighbor(tree,south);
      if ((neighbor==NULL) || (neighbor->color==white)) retval+=size;
      else if (neighbor->color==grey)
        retval+=sum_adjacent(neighbor,northwest,northeast,size);
      /* West */
      neighbor=gtequal_adj_neighbor(tree,west);
      if ((neighbor==NULL) || (neighbor->color==white)) retval+=size;
      else if (neighbor->color==grey)
        retval+=sum_adjacent(neighbor,northeast,southeast,size);
      /* North */
      neighbor=gtequal_adj_neighbor(tree,north);
      if ((neighbor==NULL) || (neighbor->color==white)) retval+=size;
      else if (neighbor->color==grey)
        retval+=sum_adjacent(neighbor,southeast,southwest,size);
      /* East */
      neighbor=gtequal_adj_neighbor(tree,east);
      if ((neighbor==NULL) || (neighbor->color==white)) retval+=size;
      else if (neighbor->color==grey)
        retval+=sum_adjacent(neighbor,southwest,northwest,size);
    }
  return retval;
```

(a) Source Codes

```
void CountTree_kernel(QuadTree tree)
{
  QuadTree nw,ne,sw,se;

  if (!tree)
     return;
  else{
       nw = tree->nw;
       CountTree_kernel(nw);
       ne = tree->ne;
       CountTree_kernel(ne);
       sw = tree->sw;
       CountTree_kernel(sw);
       se = tree->se;
       CountTree_kernel(se);
  }
}

void perimeter_kernel(QuadTree tree)
```

```
{
  QuadTree nw;

  if (!tree) return;

  nw = tree->new;

  if (tree->color==grey)
    {
      perimeter_kernel(nw);
      perimeter_kernel(tree->ne);
      perimeter_kernel(tree->sw);
      perimeter_kernel(tree->se);
    }
}
```

b) Traversal Kernel

Treeadd

```
main (int argc, char *argv[])
{
    tree_t      *root;
    int i, result = 0;


    filestuff();
    (void)dealwithargs(argc, argv);

    root = TreeAlloc (level);

    for (i = 0; i < iters; i++)
      {
        fprintf(stderr, "Iteration %d...", i);
        /* INITIALIZE PFE */
        write_kernel_id(1);
        write_root_reg(root);
        /* LDS TRAVERSAL */
        result = TreeAdd(root);
        fprintf(stderr, "done\n");
      }

    chatting("Received result of %d\n",result);
    exit(0);
}

int TreeAdd (tree_t *t)
{
  if (t == NULL)
    {
      return 0;
    }
  else
    {
      int leftval;
      int rightval;
      tree_t *tleft, *tright;
      int value;

      tleft = t->left;
      leftval = TreeAdd(tleft);
      tright = t->right;
      rightval = TreeAdd(tright);

      value = t->val;
      return leftval + rightval + value;
    }
}
```

(a) Source Codes

```
void K_TreeAdd(tree_t *t)
{
  if (t)
    {
      int leftval;
      int rightval;
      tree_t *tleft, *tright;
      int value;

      K_TreeAdd(t->left);
      K_TreeAdd(t->right);
      data_load(t->val);
    }
}
```

(b) Traversal Kernel

TSP

```
void reverse(Tree t) {
  Tree prev,back,next,tmp;

  if (!t) return;
  prev = t->prev;
  prev->next = NULL;
  t->prev = NULL;
  back = t;
  tmp = t;
  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(t);
  /* LDS TRAVERSAL */
  for (t=t->next; t; back=t,t=next) {
    next = t->next;
    t->next = back;
    back->prev = t;
    }
  tmp->next = prev;
  prev->prev = tmp;
}

static Tree conquer(Tree t) {
  Tree cycle,tmp,min,prev,next,donext;
  double mindist,test;
  double mintonext, mintoprev, ttonext, ttoprev;

  if (!t) return NULL;
  t=makelist(t);

  cycle = t;
```

```
t = t->next;
cycle->next = cycle;
cycle->prev = cycle;

for (; t; t=donext) {
  donext = t->next;
  min = cycle;
  mindist = distance(t,cycle);
  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(cycle->next);
  /* LDS TRAVERSAL */
  for (tmp=cycle->next; tmp!=cycle; tmp=tmp->next) {
    test = distance(tmp,t);
    if (test < mindist) {
      mindist = test;
      min = tmp;
      }
    }
  next = min->next;
  prev = min->prev;
  mintonext = distance(min,next);
  mintoprev = distance(min,prev);
  ttonext = distance(t,next);
  ttoprev = distance(t,prev);
  if ((ttoprev - mintoprev) < (ttonext - mintonext)) {
    prev->next = t;
    t->next = min;
    t->prev = prev;
    min->prev = t;
    }
    else {
      next->prev = t;
      t->next = next;
      min->next = t;
      t->prev = min;
      }
    }
  return cycle;
}

static Tree merge(Tree a, Tree b, Tree t) {
  Tree min,next,prev,tmp;
  double mindist,test,mintonext,mintoprev,ttonext,ttoprev;
  Tree n1,p1,n2,p2;
  double tton1,ttop1,tton2,ttop2;
  double n1ton2,n1top2,p1ton2,p1top2;
  int choice;

  min = a;
  mindist = distance(t,a);
  tmp = a;
  /* INITIALIZE PFE */
  write_kernel_id(3);
  write_reg_5(tmp);
  write_root_reg(a->next);
  /* LDS TRAVERSAL */
  for (a=a->next; a!=tmp; a=a->next) {
    test = distance(a,t);
    if (test < mindist) {
      mindist = test;
      min = a;
      }
  }
next = min->next;
prev = min->prev;
mintonext = distance(min,next);
mintoprev = distance(min,prev);
ttonext = distance(t,next);
ttoprev = distance(t,prev);
if ((ttoprev - mintoprev) < (ttonext - mintonext)) {
  p1 = prev;
  n1 = min;
  tton1 = mindist;
  ttop1 = ttoprev;
  }
else {
  p1 = min;
  n1 = next;
  ttop1 = mindist;
  tton1 = ttonext;
  }

min = b;
mindist = distance(t,b);
tmp = b;
/* INITIALIZE PFE */
write_kernel_id(3);
write_reg_5(tmp);
write_root_reg(b->next);
/* LDS TRAVERSAL */
for (b=b->next; b!=tmp; b=b->next) {
  test = distance(b,t);
  if (test < mindist) {
    mindist = test;
    min = b;
    }
  }
next = min->next;
prev = min->prev;
mintonext = distance(min,next);
mintoprev = distance(min,prev);
ttonext = distance(t,next);
ttoprev = distance(t,prev);
if ((ttoprev - mintoprev) < (ttonext - mintonext)) {
  p2 = prev;
  n2 = min;
  tton2 = mindist;
  ttop2 = ttoprev;
  }
else {
  p2 = min;
  n2 = next;
  ttop2 = mindist;
  tton2 = ttonext;
  }

n1ton2 = distance(n1,n2);
n1top2 = distance(n1,p2);
p1ton2 = distance(p1,n2);
p1top2 = distance(p1,p2);

mindist = ttop1+ttop2+n1ton2;
choice = 1;

test = ttop1+tton2+n1top2;
```

```
if (test<mindist) {
  choice = 2;
  mindist = test;
  }

test = tton1+ttop2+p1ton2;
if (test<mindist) {
  choice = 3;
  mindist = test;
  }

test = tton1+tton2+p1top2;
if (test<mindist) choice = 4;

switch (choice) {
  case 1:
    reverse(n2);
    p1->next = t;
    t->prev = p1;
    t->next = p2;
    p2->prev = t;
    n2->next = n1;
    n1->prev = n2;
    break;
  case 2:
    p1->next = t;
    t->prev = p1;
    t->next = n2;
    n2->prev = t;
    p2->next = n1;
    n1->prev = p2;
    break;
  case 3:
      p2->next = t;
      t->prev = p2;
      t->next = n1;
      n1->prev = t;
      p1->next = n2;
      n2->prev = p1;
      break;
    case 4:
      reverse(n1);
      n1->next = t;
      t->prev = n1;
      t->next = n2;
      n2->prev = t;
      p2->next = p1;
      p1->prev = p2;
      break;
    }
  return t;
}
```

(a) Source Codes

```
void reverse_kernel(Tree t) {
  Tree prev,back,next,tmp;

  if (!t) return;
  for (t=t->next; t;t=next) {
    next = t->next;
    }
}
```

```
void conquer_kernel(Tree t) {
  Tree next,donext;

  for (t=t->next; t; t=donext) {
    donext = t->next;
    data_load(t->x);
    data_load(t->y);
  }
}


void merge_kernel(Tree a, Tree stop) {

  for (a=a->next; a!=stop; a=a->next) {
    data_load(a->x);
    data_load(a->y);
  }
}
```

(b) Traversal Kernel

Bisort

```
int Bisort(HANDLE *root, int spr_val, int dir)
{
  if (root->left == NIL)
    {
     if ((root->value > spr_val) ^ dir)
        {
          int val;
          val = spr_val;
          spr_val = root->value;
          root->value =val;
        }
    }
  else
    {
      root->value=Bisort(root->left,root->value,dir);
      spr_val=Bisort(root->right,spr_val,!dir);
      /* INITIALIZE PFE */
      write_kernel_id(1);
      write_root_reg(root);
      /* LDS TRAVERSAL */
      spr_val=Bimerge(root,spr_val,dir);
    }
  return spr_val;
}

int Bimerge(HANDLE *root, int spr_val, int dir)
{
  int rightexchange, elementexchange;
  HANDLE *pl, *pr;
```

```
rightexchange = ((root->value > spr_val) ^ dir);
if (rightexchange)
  {
    int temp;
    temp = root->value;
    root->value = spr_val;
    spr_val = temp;
  }

pl = root->left;
pr = root->right;

while (pl != NIL)
  {
    elementexchange = ((pl->value > pr->value) ^ dir);
    if (rightexchange)
    {
        if (elementexchange)
        {
            SwapVal(pl,pr);
            SwapRight(pl,pr);
            pl = pl->left;
            pr = pr->left;
        }
        else
        {
            pl = pl->right;
            pr = pr->right;
        }
    }
```

```
      else
      {
          if (elementexchange)
          {
              SwapVal(pl,pr);
              SwapLeft(pl,pr);
              pl = pl->right;
              pr = pr->right;
          }
          else
          {
              pl = pl->left;
              pr = pr->left;
          }
      }
     }

   if (root->left != NIL)
    {
      root->value=Bimerge(root->left,root->value,dir);
      spr_val=Bimerge(root->right,spr_val,dir);
    }
  return spr_val;
}
```

(a) Source Codes

```
void kernel(HANDLE *root)
{
  HANDLE *left, *right;

  left = root->left;
  right = root->right;
  data_load(root->value)
  kernel(left);
  kernel(right);
}
```

(b) Traversal Kernel

Voronoi

```
QUAD_EDGE build_delaunay_triangulation
            (VERTEX_PTR tree, VERTEX_PTR extra)
{
    EDGE_PAIR retval;
    /* INITIALIZE PFE */
    write_kernel_id(1);
    write_root_reg(tree);
    /* LDS TRAVERSAL */
    retval=build_delaunay(tree,extra);
    return retval.left;
}

EDGE_PAIR build_delaunay(VERTEX_PTR tree,extra)
{
    QUAD_EDGE a,b,c,ldo,rdi,ldi,rdo;
    EDGE_PAIR retval;
    register VERTEX_PTR maxx, minx;
    VERTEX_PTR s1, s2, s3;

    EDGE_PAIR delleft, delright;

    if (tree && tree->right)
      {
        minx = get_low(tree); maxx = extra;
        delright = build_delaunay(tree->right,extra);
        delleft = build_delaunay(tree->left, tree);
        ldo = delleft.left; ldi=delleft.right;
        rdi=delright.left; rdo=delright.right;
        retval=do_merge(ldo, ldi, rdi, rdo);
        ldo = retval.left;
    rdo = retval.right;
    while (orig(ldo) != minx)
        {NOTEST(); ldo = rprev(ldo); RETEST();}
    while (orig(rdo) != maxx)
        {NOTEST(); rdo = lprev(rdo); RETEST();}
    retval.left = ldo;
    retval.right = rdo;
}
else if (!tree)
  {
    printf("ERROR: Only 1 point!\n");
   exit(-1);
  }
else if (!tree->left) {
    a = makeedge(tree, extra);
    retval.left =  a;
    retval.right = sym(a);
}
else {
    s1 = tree->left;
    s2 = tree;
    s3 = extra;
    a = makeedge(s1, s2);
    b = makeedge(s2, s3);
    splice(sym(a), b);
    c = connect_left(b, a);
    if (ccw(s1, s3, s2)) {
        retval.left = sym(c);
        retval.right = c;
    }
```

```
else {
    retval.left =  a;
    retval.right = sym(b);
    if (!ccw(s1, s2, s3)) deleteedge(c);    /* colinear */
        }
    }
    return retval;
}
```

(a) Source Codes

```
void kernel(VERTEX_PTR tree)
{
  if (tree)
    {
      kernel(tree->right);
      kernel(tree->left);
    }
}
```

(b) Taversal Kernel

Rayshade

```
int ListIntersect(List *list, Ray *ray, HitList *hitlist, Float mindist,
                  Float maxdist)
{
  Geom *objlist;
  Vector vtmp;
  Float s;
  int hit;

  hit = FALSE;
  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(list->unbounded);
  /* LDS TRAVERSAL */
  for (objlist = list->unbounded; objlist ; objlist = objlist->next) {
    if (intersect(objlist, ray, hitlist, mindist, maxdist))
         hit = TRUE;
  }

  s = *maxdist;
  VecAddScaled(ray->pos, mindist, ray->dir, &vtmp);
  if (OutOfBounds(&vtmp, list->bounds) &&
     !BoundsIntersect(ray, list->bounds, mindist, &s))
       return hit;

  /* INITIALIZE PFE */
  write_kernel_id(1);
  write_root_reg(list->list);
  /* LDS TRAVERSAL */
  for (objlist = list->list; objlist ; objlist = objlist->next) {
     if (intersect(objlist, ray, hitlist, mindist, maxdist))
```

```
       hit = TRUE;
  }
  return hit;
}
int intersect(Geom *obj, Ray *ray, HitList *hitlist, Float mindist,
              Float *maxdist)
{
  Ray newray;
  Vector vtmp;
  Trans *curtrans;
  Float distfact, nmindist, nmaxdist;

  if (obj->methods->checkbounds) {
     VecAddScaled(ray->pos, mindist, ray->dir, &vtmp);
     if (OutOfBounds(&vtmp, obj->bounds)) {
       nmaxdist = *maxdist;
       BVTests++;
       if (!BoundsIntersect(ray, obj->bounds, mindist,&nmaxdist))
        return FALSE;
       }
   }

   newray = *ray;
   nmindist = mindist;
   nmaxdist = *maxdist;

   if (obj->trans != (Trans *)0) {
      if (obj->animtrans && !equal(obj->timenow, ray->time)) {
          TransResolveAssoc(obj->trans);
       }
       distfact = 1.;
       for (curtrans = obj->transtail; curtrans;
            curtrans = curtrans->prev)
           distfact *= RayTransform(&newray, &curtrans->itrans);
       nmindist *= distfact;
       nmaxdist *= distfact;
      }
      obj->timenow = ray->time;
      if (IsAggregate(obj)) {
        if (!(*obj->methods->intersect)
              (obj->obj, &newray, hitlist, nmindist, &nmaxdist))
            return FALSE;
        } else {
          if (!(*obj->methods->intersect)
                (obj->obj, &newray, nmindist, &nmaxdist))
            return FALSE;
            hitlist->nodes = 0;
        }
        AddToHitList(hitlist, &newray, nmindist, nmaxdist, obj);
        if (obj->trans != (Trans *)0)
            *maxdist = nmaxdist / distfact;
        else
            *maxdist = nmaxdist;

        return TRUE;
}
```

(a) Source Codes

```
void kernel(Geom *obj)
{
  struct Methods *method;
  Geom *next;

  while (obj)
```

```
    {
      next = obj->next;
      method = obj->methods;
      if (method->checkbounds)
        {
          data_load (obj->bounds[0][0]);
        }
      if (obj->trans != 0)
        {
          data_load(obj->animtrans);
          data_load(obj->timenow);
          data_load(obj->transtail);
        }
      data_load(method->normal);
      data_load(method->intersect;
      data_load(obj->obj);
      obj = next;
    }
}
```

(b) Traversal Kernel

Bibliography

[1] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowics. April: a processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[2] T. Alexander and G. Kedem. Distributed predictive cache design for high performance memory system. In *Proceedings of the 2th International Symposium on High-Performance Computer Architecture*, February 1996.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Prceedindings of International Conference on Supercomputing*, pages 1–6, June 1990.

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

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

[6] G. A. Brent. *Using Programs Structure to Achieve Prefetching For Cache Memory.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1987.

[7] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors-the simplescalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

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

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

[10] M. J. Charney and A. P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feburary

1995.

[11] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[12] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceeding of Micro-computing 24*, pages 69–73, November 1991.

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

[14] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious struture layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[15] J. D. Collins, H. Wang, D. M. Tullsen, H. J. Christopher, Y. F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, June 2001.

[16] S. P. Crago, A. Despain, J. Gaudiot, M. Makhija, W. Ro, and A. Sricastava. A high-performance, hierarchical decoupled architecture. In *Proceedings of the Memory Access Decoupling for SuperScalar and Multiple Issue Architecture Workship*, October 2000.

[17] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1999. San Fransisco.

[18] C. Ding and K. Kenndy. Improving cache performance in dynamic applications through data and computation organization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, May 1999.

[19] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the ACM International Conference on Supercomputing*, pages 176–186, May 1997.

[20] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

[21] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, December 1992.

[22] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of SIGPLAN Symposium on Compiler Construction*, pages 11–16, June 1986.

[23] Peter N. Glaskowsky. Pentium 4 (partially) reviewed. *Microprocessor Report*, August 2000.

[24] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-detected data prefething in multiprocessor with memory hierarchy. In *Prceedindings of International Conference on Supercomputing*, 1990.

[25] Linley Gwennap. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*, October 1998.

[26] R. H. Halstead and T. Fujita. Masa: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, May 1988.

[27] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1996. San Fransisco.

[28] C. J. Hughes. Prefetching linked data structures in systems with merged dram-logic, masters thesis. Technical report, Duke University Computer Science Department, May 1998.

[29] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[30] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[31] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. Flexram: Toward an advanced intelligent memory system. In *Proceedings of the 1999 International Conference on Computer Design*, pages 192–201, October 1999.

[32] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of Sixth Symposium High-Performance Computer Architecture*, pages 206–217, January 1999.

[33] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, pages 34–36, April 1999.

[34] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, 1991.

[35] C. Kolb. The rayshade user's guide. In *http://graphics.stanford.edu/- cek/-rayshade.*

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

[37] M. Lam. Software pipelining: an efficient scheduling technique for vliw machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[38] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: A case study. In *IEEE Computer*, pages 15–26, 1994.

[39] Steve Leibson. Xscale (strongarm-2) muscles in. *Microprocessor Report*, September 2000.

[40] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: Software prefeteching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, June 1995.

[41] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the*

28th Annual International Symposium on Computer Architecture, pages 40–51, June 2001.

[42] C.-K. Luk and T. C. Mowry. 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)*, pages 222–233, October 1996.

[43] S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric program. In *Proceedings of the 10th International Conference on Supercomputing*, pages 133–139, May 1996.

[44] T. C. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessor. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[45] T. C. Mowry, M. S. Lam, and A. Gupta. 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*, pages 62–73, October 1992.

[46] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: a computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, June 1998.

[47] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[48] D. Patterson, T. Andreson, N. Cardwell, R. Fromm, K. Keaton, C. Kazyrakis, R. Thomas, and K. Yellick. A case for intelligent ram. *IEEE Micro*, pages 34–44, April 1997.

[49] S. Pinter and A. Yoaz. A hardware-based data prefetching technique for superscalar processor. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 214–225, December 1996.

[50] A. R. Pleszkun and E. S. Davidson. Strutured memory access architecture. In *Prceedindings of International Conference on Parallel Processing*, pages 461–471, 1983.

[51] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, 1989.

[52] S. Przybylski. Embedded drams: Today and toward system-level intergration. In *Tutorial with the Annual International Symposium on Computer Architecture*, May 1997.

[53] A. Roger, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Sytems*, 17(2), March 1995.

[54] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 115–126, October 1998.

[55] A. Roth and G. Sohi. New mehtods for exploiting program structure and behavior in computer architecture. In *Proceedings of International Workshop on Innovative Architecture*, October 1998.

[56] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[57] A. Roth and G. Sohi. A simple and efficient implementation of squash re-use. In *Proceedings of the 33st Annual International Symposium on Microarchitecture*, pages 223–234, December 2000.

[58] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of 7th Symposium High-Performance Computer Architecture*, pages 134–143, January 2001.

[59] I. Sklenar. Prefetch unit for vector operations on scalar computers. *Computer Architecture News*, 20(4):31–37, September 1992.

[60] A. Smith. Cache memories. *Computing Surveys*, 14(4):473–530, September 1982.

[61] B. Smith. Architecture and applications of the hep multiprocessor computer system. In *Proceedings of the Int. Soc. for Opt. Engr*, pages 241–248, 1982.

[62] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, 1982.

[63] G. Sohi. Instruction issue logic for high performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[64] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 257–268, November 2000.

[65] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[66] S. VanderWiel and D. J. Lilja. A compiler-assisted data prefetch controller. Technical Report ARCTiC-99-05, Department of Electrical and Computer Engineering, University of Minnesota, May 1999.

[67] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[68] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[69] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–200, June 1995.

[70] C. B. Zilles and G. Sohi. Execution-base prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, June 2001.

Biography

Chia-Lin Yang received the B.S. degree from the National Taiwan Normal University in 1989 and the M.S. degree from the University of Texas at Austin in 1992. In 1993, She joined VLSI Technology Inc. (now Philips Semiconductors) as a software engineer. From 1995 to 2001 she attended Duke University and received her Ph.D. degree in Computer Science. She will join the Department of Computer Science and Information Engineering at the National Taiwan University in Taiwan. Yang is the recipient of a 2000-01 Intel Foundation Graduate Fellowship Award and a member of ACM.