IPU/LTB: A METHOD FOR REDUCING EFFECTIVE MEMORY LATENCY

A Thesis
Presented to
The Academic Faculty

by

C. Reid Harmon Jr.

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
November 2003
IFU/LTB: A METHOD FOR REDUCING EFFECTIVE MEMORY LATENCY

Approved by:

Ken MacKenzie, Advisor

Bill Appelbee

Helen-Hain S. Leu

Sanjoli Pande

Kishore Ramachandran

Date Approved 11-20-2003
Thanks go to the faculty and students at U.C. Berkeley and McGill University who developed the original DLX and SuperDLX simulators, respectively; to Maurizio Vitale, who gave me the introduction to PostScript which paved the way for the analysis utility used to verify cycle-by-cycle operation of the simulator visually, and who gave me many helpful suggestions while making modifications to the original SuperDLX simulator; to Paul May, who tested the simulator and provided feedback on its usability; to Scott Wills, who wrote and supplied the cachexim simulator, and who kept my ideas grounded in reality; and to Raja Das, who read over the early work carefully and who posed many thought-provoking questions.

Special thanks to my reading committee; to Bii Appelbe, who took me under his wing, whose research in future branches necessitated the SuperDLX simulator, whose own ideas on prefetching are the kernel of those presented in this work, and who provided the direction for the research; to Kishore Ramachandran, for keeping the research honest on the type of benchmarks used and the parameters for the experiments; and to Hsien-Hsin "Sean" Lee and Santosh Pande, for agreeing to read this research on short notice but who nevertheless read with sharp eyes and penetrating insight. Each one of you has improved the quality of the research by making sure that certain details not be neglected.

Thanks especially to Ken Mackenzie, for taking on this research as a replacement advisor and for getting familiar with it so quickly; for keeping the research above all else fair and honest; for asking the relevant questions and keeping me focused on the important answers; for being a wonderful sounding board and for helping me to gel my thoughts into words; for aiding in the organization of this work; and for being patient over the time it has taken for this research to come to fruition.

Most of all, thanks to my friends for their encouragement; to my boss, Dr. E. Eugene Baillie, for allowing me the flexible hours required to complete this work; and my family,
for their constant support, encouragement, faith, and love. To my wife Angie and my sons Chase and Matthew, thank you; without you there would have been no reason for me to see this through.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................................................ iii

LIST OF TABLES .................................................................................................................................. viii

LIST OF FIGURES .................................................................................................................................. ix

ABSTRACT ............................................................................................................................................... xii

I OVERVIEW ............................................................................................................................................... 1

II BACKGROUND ....................................................................................................................................... 6

2.1 Latency Reduction Techniques ........................................................................................................ 7

2.1.1 Compiler Optimizations ................................................................................................................ 8

2.1.2 Software Solutions .......................................................................................................................... 9

2.1.3 Hardware Solutions ....................................................................................................................... 10

2.2 Latency Masking Techniques ........................................................................................................... 10

2.2.1 Compiler Optimizations ................................................................................................................ 11

2.2.2 Software Solutions .......................................................................................................................... 12

2.2.3 Hardware Solutions .......................................................................................................................... 13

2.2.4 Combined Approaches .................................................................................................................... 14

2.3 Prefetching ......................................................................................................................................... 15

2.3.1 Speculation Techniques .................................................................................................................. 16

2.3.2 Problems with Prefetching ............................................................................................................. 18

2.3.3 Variants of Prefetching .................................................................................................................... 18

2.3.4 Prefetching for Shared-Memory Architectures ............................................................................. 20

2.4 Related Work ..................................................................................................................................... 21

2.4.1 Software Approaches ..................................................................................................................... 21

2.4.2 Hardware Approaches .................................................................................................................... 28

2.4.3 Combined Approaches .................................................................................................................... 40

2.5 Proposal ............................................................................................................................................. 41

III HARDWARE SUPPORT ...................................................................................................................... 43

3.1 IPU .................................................................................................................................................... 43

3.1.1 Calculating ahead .......................................................................................................................... 46
LIST OF TABLES

Table 1  Simulation parameters which differ from the X10000 ............... 64
Table 2  Benchmark configurations ........................................ 67
Table 3  IPU Statistics on I-Prefetching Candidate Benchmarks ............... 84
LIST OF FIGURES

Figure 1 Interdependence of the processor and memory ........................................ 7
Figure 2 Increased memory latency masking through prefetching ............................... 15
Figure 3 Instruction Prefetch Unit block diagram ................................................... 41
Figure 4 Example code .......................................................................................... 47
Figure 5 Example execution state for uninformed instruction prefetching ................. 48
Figure 6 Example stack representations for uninformed instruction prefetching ......... 48
Figure 7 Example execution state for informed instruction prefetching .................... 50
Figure 8 Example stack representations for informed instruction prefetching ......... 50
Figure 9 Algorithm for incremental instruction prefetching ................................... 51
Figure 10 Example execution state for incremental instruction prefetching ............. 52
Figure 11 Separate Prefetch Cache (PFC) configuration ........................................... 53
Figure 12 Load Target Buffer block diagram ......................................................... 54
Figure 13 Bit encoding for MIPS indirect loads ...................................................... 57
Figure 14 Bit requirements of indirect load offsets of SPEC95 benchmarks ............. 57
Figure 15 Compositions for loads with large offsets ................................................ 58
Figure 16 An example of an innermost loop with multiple traces .......................... 59
Figure 17 Sample output for daepy ...................................................................... 69
Figure 18 Composition of execution time for the int SPEC95 benchmarks .......... 75
Figure 19 Composition of execution time for the fp SPEC95 benchmarks ............ 76
Figure 20 IPC speedups for uninformed instruction prefetching ............................... 77
Figure 21 Fetch cycle recovery rates for uninformed instruction prefetching .......... 78
Figure 22 IPC speedups for informed instruction prefetching .................................. 79
Figure 23 Fetch cycle recovery rates for informed instruction prefetching ............. 80
Figure 24 IPC speedups for incremental instruction prefetching ............................ 81
Figure 25 Fetch cycle recovery rates for incremental instruction prefetching ........... 81
Figure 26 IPC speedups for various instruction prefetching configurations ............. 82
Figure 27 IPC speedups versus values of K and N .................................................. 85
Figure 28 Candidates exhibiting data prefetching slowdowns ............................... 86
Figure 29 Candidates exhibiting data prefetching speedups .................................. 87
Figure 30 A histogram of calculated values of N for the SPEC95 benchmarks.

Figure 31 Candidates exhibiting data prefetching slowdowns for static N and K.

Figure 32 Candidates exhibiting data prefetching speedups for static N and K.

Figure 33 Prefetching vs add. CPU hardware - int benchmarks.

Figure 34 Prefetching vs add. CPU hardware - fp benchmarks.

Figure 35 Prefetching vs add. CPU hardware, fully 4-way - int benchmarks.

Figure 36 Prefetching vs add. CPU hardware, fully 4-way - fp benchmarks.

Figure 37 IPU versus Reinman - int benchmarks.

Figure 38 IPU versus Reinman - fp benchmarks.

Figure 39 L2B versus Baer/Chen - int benchmarks.

Figure 40 L2B versus Baer/Chen - fp benchmarks.

Figure 41 Chen sensitivity to N and K.

Figure 42 IPU/LTB versus Reinman/Baer/Chen - int benchmarks.

Figure 43 IPU/LTB versus Reinman/Baer/Chen - fp benchmarks.

Figure 44 Speedup versus degree of pipelining.

Figure 45 Speedup versus degree of pipelining - reduced L2 line.

Figure 46 Speedup versus main memory latency - int benchmarks.

Figure 47 Speedup versus main memory latency - fp benchmarks.

Figure 48 Speedup versus main memory bus width - int benchmarks.

Figure 49 Speedup versus main memory bus width - fp benchmarks.

Figure 50 Speedup versus L1 cache size - int benchmarks.

Figure 51 Speedup versus L1 cache size - fp benchmarks.

Figure 52 Cache behavior for benchmark appl.

Figure 53 Cache behavior for benchmark apsc.

Figure 54 Cache behavior for benchmark compress.

Figure 55 Cache behavior for benchmark fppp.

Figure 56 Cache behavior for benchmark gcc.

Figure 57 Cache behavior for benchmark go.

Figure 58 Cache behavior for benchmark hydrazd.

Figure 59 Cache behavior for benchmark ljpeg.

Figure 60 Cache behavior for benchmark ll.
Processing speed continues to outperform the memory subsystem, enabling data-heavy research which would have been unthinkable even five years ago. The large applications run on modern processors, coupled with this discrepancy between the processor and memory speeds, have caused the memory subsystem to become even more of a system bottleneck. While innovative cache designs and prefetching methods address this problem, they are designed for either data prefetching or instruction prefetching, not allowing both types to co-exist, and therefore favoring certain types of applications over others.

While instruction prefetching strategies have employed the branch predictor to predict the instruction cache lines needed, these strategies fail when used in conjunction with increasing memory latencies, and they waste time in inner loops which fit in the L-cache. Data prefetching can suffer from this problem to a lesser degree when it is tied to a branch-based lookahead. This research proposes new hardware units to perform instruction and data prefetching. The IPU, or instruction prefetching unit, uses a scheme which follows only interprocedural control-flow, ignoring branches, and using a hardware return address stack. The LTB, or load target buffer, is a 4-way associative buffer based on load instruction address, and which makes a prediction when load addresses are calculated in hardware. The access pattern is predicted using stride prediction, and the load target prefetched is the one $N$ strides ahead, where $N$ is encoded into the load instruction. The compiler algorithm to place this information is discussed.

Experiments were conducted via a hardware-level simulator called SuperDLX, a superscalar simulator developed by McGill University based on the theoretical DLX architecture proposed by Hennessy and Patterson. A range of dynamic instructions were chosen from each of the SPEC95 benchmarks for steady-state or whole cycles of periodic memory behavior.
The experiments yielded several clear trends:

1. Inter-procedural control flow is better suited to instruction prefetching than branch-based control flow in modern superscalar processors. (See Section 4.7.)

2. A simple incremental scheme to control instruction prefetching distance is more effective than protected-mode caches or separate prefetch caches, and it is nearly as effective as more elaborate schemes based on the call stack. (See Section 4.3.)

3. Modern memory system latencies require data prefetching more than one iteration ahead in inner loops. (See Section 4.4.)

4. Prefetch distance encoding performs better than lookahead mechanisms for stride-based data prefetching in modern superscalar processors, due to this need of larger distances. (See Section 4.7.)

5. Prefetching is cost effective. (See Section 4.6.)

6. The IPU/LTB approach holds up well against a trend of an increasing application/cache ratio. (See Section 4.8.)
CHAPTER I

OVERVIEW

The past decade has witnessed many advances to computing speed. Processor clock rates have increased two orders of magnitude, and processors have been designed to capitalize on instruction level parallelism (ILP). This type of parallelism has its limits, however; true dependence chains wind through the dynamic instruction trace. When all available ILP has been exploited, and when the branch prediction mechanism can deliver a continuous stream of correct instruction addresses, then the only gains left for such a processor is to attack the true dependence chains of the dynamic trace.

The factors affecting the execution of the primary dynamic trace include each instruction fetch latency and each instruction execution latency. Poor I-cache performance therefore directly restrains execution speed. Similarly, load latencies during D-cache misses are among the longest instruction execution latencies. In addition, the relative latency required in servicing a miss is increasing, since the rate of increase in memory speeds has not matched the rate of increase in processor clock rates. All of this points to an increase in the capability of the memory system as a primary source of program execution speedup.

The memory system can be augmented in many ways: by reducing the number of I-cache or D-cache misses; by reducing the miss or hit wait latency; by allowing the overlap of multiple memory operations; and by prioritizing operations to minimize overall execution time. Of these, the most interesting are the ones which assume main memory and its bus as a constant and which seek to improve program execution by changing the cache operations and usage.

These methods can be broadly classified into one of two categories: latency reduction techniques and latency masking (or hiding) techniques. Latency reduction refers to methods which reduce the total amount of latency, as measured by the processor as the sum of the

1 Hit wait latency is latency incurred by waiting on a previously requested cache line which has not arrived
Latency masking techniques are those which aim to further overlap memory latency with useful work. By nature these techniques are speculative, meaning that they sometimes require action when it cannot yet be determined whether or not the action is correct for the circumstances. Latency masking techniques do not alter the total latency, but they do break the true dependence chain to allow memory operations in the chain to occur at the same time as the execution of instructions which depend on them or instructions on which the memory operations depend. This therefore increases the amount of parallelism being exploited, when the speculation is correct.

Latency reduction and latency masking techniques can also be broadly classified into one of three categories: compiler optimizations, software solutions, and hardware solutions. Compiler optimizations are techniques which can be applied by a compiler and which do not require change to the hardware. Similarly, hardware solutions are those which address cache-side memory augmentation in hardware, and which do not require recompilation to function. Software solutions are those in which both hardware support and compiler technology are required.

Between latency reduction and latency masking techniques, latency masking techniques are more interesting in general because of their speculative nature. And of these techniques, the most interesting subcategory is the most speculative set of approaches, collectively called prefetching. Prefetching techniques span across the classifications of compiler optimizations, software solutions, and hardware solutions; they work by predicting what information will be needed in the cache in the future and requesting this information early enough that the requesting hardware unit does not incur miss latency or bus wait latency. The total latency does not change because the miss latency is still incurred by the prefetching unit. In fact, if the prefetched information is not needed, the total latency will increase; and if the prefetched information replaced needed information in the cache, then yet another memory operation will be required to bring the needed information back into the cache. These
potential problems with prefetching only add to its allure as a field of study. Questions remain open about how to perform prefetching well and under what conditions.

This document is a proposal of a novel form of hardware prefetching, one which separates instruction prefetching and data prefetching and treats each according to the way in which memory is accessed for each. The instruction prefetching is performed by the instruction prefetching unit (IPU). The IPU sequentially fetches I-cache lines and predecodes them, searching for instructions which change the program's call stack. If the target address of the instruction is determinate without instruction execution, the IPU follows the target and continues sequential prefetching from the new point. The IPU maintains its own call stack in hardware, and this approach requires that the Fetch Unit use its own hardware-based call stack, often called a return address stack (RAS). When the Fetch Unit encounters an I-cache miss, the IPU takes this to mean that it is either following the wrong path or that it is displacing needed I-cache lines. Therefore, it resets itself to mirror the state of the Fetch Unit and begins again. In an attempt to limit the latter case, the IPU maintains a counter of how many words ahead of the Fetch Unit it thinks is is, and only proceeds if this counter, called the ahead value, is less than or equal to a fixed threshold value. The IPU maintains the ahead value by adding the number of instructions it progresses and subtracting the number of instruction the Fetch Unit progresses each cycle. The progression is measured by the number of instructions inside procedures, but not across procedures. In other words, address jumps caused by branches do affect this value, but address jumps caused by procedure calls or returns do not.

The data prefetching component is a software technique that avoids placement and execution of prefetching instructions. Instead, loads are classified by whether they are prefetching loads and what the desired prefetching distance is, as measured in loop iterations. This technique is implemented by reserving up to three bits in the instruction to indicate the load's prefetching status. When three bits are used, a non-prefetching load and prefetching loads with seven different distances are possible. The actual distances used are fixed in a hardware table, and are indexed by the reserved bits. Unlike lookahead mechanisms, the prefetching does not occur for that instance of the load instruction, but instead for N
iterations ahead, where \( N \) is the value referred to in the hardware table. Prediction of this
address is the standard three-stride prediction mechanism: a table called the LTB is kept of
recent load instructions, and for each load instruction the most recent load target address
\( A \), and the three most recent strides are stored. When two of the three recent strides match,
the next address is predicted to be \( A + S \), where \( S \) is the common stride. The LTB predicts
the \( N \)’th address to be \( A + N \times S \). The LTB is also different from look-ahead mechanisms in
that the prefetching trigger is the calculation of a new load address, not the appearance of
the load in the instruction window. In other words, in loop iteration \( i \), a request for memory
item \( i \) will appear concurrently with a prefetch request for memory item \( i + N \). The values
of \( N \) must be calculated at compile time in order to be fixed in the load instructions. The
value used is the smallest value in the hardware table which is larger than \( \frac{E_{\text{loop}}}{M} \), where
\( E_{\text{loop}} \) is the computed average execution time of the loop and \( M \) is the total latency of
retrieving a D-cache line from main memory, assuming no waiting on previous requests.
A final enhancement was made to traditional stride-based prefetching in that prefetching
would wait until at least \( K \) load addresses had been calculated for an instruction before the
first prefetch request could be issued, where \( K \) is a value fixed in hardware.

This method was tested by simulating the execution of the SPEC95 benchmarks on
a superscalar DLX architecture. Some of the benchmarks required too much processing
time to simulate them in their entirety for all the experiments performed. In order to
to obtain a picture of the benchmarks’ memory performance over time, each benchmark was
simulated in its entirety, and memory statistics were recorded at regular intervals. These
statistics were then used to select a range of dynamic instruction counts to be used for the
experiments. In each case at least 100 million instructions were chosen after the point at
which the benchmarks had settled into steady-state behavior. Some benchmarks exhibited
periodic behavior, and for those benchmarks whole periods were selected of not less than
100 million instructions total (see Appendix A).

The experiments revealed first that 2048 was a suitable number for the instruction
prefetching’s threshold. Further experiments showed that if data prefetching were employed
without encoding bits into the instruction that the greatest yield came when \( N \) was 2 and
K was 3. When three bits were used, however, the results were better. The seven distances represented by the three encoded bits were 1, 2, 3, 6, 9, 12, and 19.

The experiments also revealed an interesting characteristic of the benchmarks; most benefited from instruction prefetching or data prefetching, but none benefited from both. Fortunately, when one type of prefetching worked and the other did not, enabling these both did not adversely affect the working method.

Finally, the experiments revealed several clear trends:

1. Inter-procedural control flow is better suited to instruction prefetching than branch-based control flow in modern superscalar processors. This is demonstrated in a direct comparison to a branch-based scheme in Section 4.7.

2. A simple incremental scheme to control instruction prefetching distance is more effective than protected-mode caches or separate prefetch caches, and it is nearly as effective as more elaborate schemes based on the call stack. This is demonstrated in a comparison among various flavors of inter-procedural instruction prefetching in Section 4.3.

3. Modern memory system latencies require data prefetching more than one iteration ahead in inner loops. This is shown in a study in Section 4.4 in which the prefetching distance is varied.

4. Prefetch distance encoding performs better than lookahead mechanisms for stride-based data prefetching in modern superscalar processors, due to this need of larger distances. This is demonstrated in Section 4.7 by directly comparing the LTB against Basr and Chen's lookahead method.

5. Prefetching is cost effective. This is demonstrated in a comparison of IPU/LTB to adding more functional units, in Section 4.6.

6. The IPU/LTB approach holds up well against a trend of an increasing application/cache ratio. This is shown in Section 4.8, in which the performance of IPU/LTB is evaluated with respect to changing machine parameters.
CHAPTER II

BACKGROUND

Both processor speeds and memory speeds have enjoyed a near-exponential growth over the last 20 years. The growth factor for processor speeds, however, has been higher than that of memory speeds. For this reason, the relative memory access time has been increasing as well, and is two orders of magnitude higher than it was 20 years ago. Memory access is therefore becoming a primary bottleneck in program execution.

This bottleneck can be alleviated, however, and without increasing memory bandwidth. Figure 1 illustrates a parallelism that may be exploited, one between the processing unit and the memory system. Each of the two systems cooperate to complete a task, and each depends on information from the other to proceed. The processing unit provides memory addresses to the memory system, and the memory system provides data to the processing unit. This interdependence prevents the completion of the work in the minimum amount of time.

When the processing unit can exploit instruction-level parallelism (ILP), it can continue to process instructions which do not depend on outstanding loads. In this way, it reduces the amount of time it spends waiting on the result of the load, by partially masking the load latency. Unmasked latency, however, reveals inefficiency in the completion of the task at hand, since parallel operations are being executed sequentially, and as memory latencies increase, the amount of unmasked latency incurred also increases, which in turn increases the overall execution time of the task.

The idea of memory latency masking is only pure for a single-issue processor with non-blocking cache access, but it is possible to generalize this phenomenon to multiple-issue processors as well. For example, assume a fully 4-way superscalar processor executing an inner loop with two parallel data dependence chains. If the execution of dependence chain $A$ is delayed due to a cache miss, it can be masked by the execution of chain $B$ only if the
execution time of $B$ is longer than $A$ by at least the load latency. For additional chains $C$ and $D$, only one chain must have a sufficiently larger execution time than $A$. And if there are more than four chains, the load latency incurred by $A$ tends to be amortized across the four pipelines. In other words, the effect is that one-fourth of the load latency is unmasked.

In practice, however, the wide-issue capacity of modern processors is now outpacing the available ILP. One study [134] estimates the available ILP for most programs as 5, even under ideal execution conditions. This means that the ability of the processor to mask load latency with computation has become severely limited, and addressing the long latency problem is a fertile area of research for speeding up program execution.

2.1 Latency Reduction Techniques

Some solutions to the long latency problem are latency reduction techniques. These techniques attempt to reduce the total amount of real latency incurred in the execution of a task. They tend to fall into one of three categories. The first of these is through the use of compiler technology alone; these methods are generally referred to as compiler optimizations. The second category is software solutions, which means compiler technology
supported by hardware. The hardware exists only to implement the instructions executed and is therefore secondary to the compiler technology which generates them. The third category is called hardware solutions, which means improvements gained through specialized hardware without requiring changes in the code.

2.1.1 Compiler Optimizations

Since memory latency is a function of hardware, the only way a compiler can reduce the total real latency incurred is to eliminate unnecessary memory operations. This is a beneficial side-effect of common subexpression elimination; the compiler removes duplicate calculations, and thus reduces the number of fetches required to complete the program and often occurrences of loads and stores required for the calculation. Similarly, intelligent register allocation removes many memory data operations which reduces the number of these as well as fetches required.

Additionally, rearranging memory operations can reduce conflict cache misses and therefore reduce the number of accesses to main memory. This is the basis behind sophisticated scheduling and blocking techniques [73, 101] which seek to improve cache performance.

The same benefit may be applied to instructions by rearranging the static layout of the executed code to better agree with the predicted dynamic path [139]. Infrequently executed code can be mapped to a small area of the cache, leaving the remaining contiguous basic blocks to be reassembled into one long static trace in memory.

Cache reuse of data can be enhanced by manipulating the original algorithm at a high level, thus changing the data access pattern [22]. It is usually necessary for the compiler to have detailed information on the organization and policies of the memory system in order to use these techniques successfully. Such optimizations include the loop reconstruction techniques permutation, fusion, distribution, skewing, and reversal. Tilting is another technique for rearranging access for nested loops; the iteration space is partitioned into tiles in which each tile internally exhibits a high degree of locality. Each tile, then, is executed
separately\textsuperscript{4}. Without such an optimization, the program as written may traverse the iteration space from tile to tile, experiencing locality only for a brief time before moving on to a different region of memory, from which it must load data into the cache again.

To be complete, however, a compiler algorithm must consider not only the large statically allocated arrays of data but also the stack, heap, and constants. A temporal relationship graph \cite{19} reveals how items in these places may cause cache conflicts and thus provide information about how best to place the items in memory to reduce these conflicts. Additionally, CME's \cite{48} are a technique for mathematically calculating cache miss rates, and they allow a compiler to determine the optimal combination of applied optimizations.

2.1.2 Software Solutions

When data is cached on the basis of virtual address, simply remapping the physical to virtual addresses can alleviate conflict of two or more data spaces competing for residency in a direct-mapped cache \cite{13}. This technique requires a piece of hardware called a Cache Miss Lookaside (CML) for tracking high conflict spaces, as well as the software in the operating system to examine the CML and remap virtual memory based on the findings. A similar idea targets just compiler-generated spill code \cite{34} with a small compiler-controlled memory to alleviate conflicts between register spills and the regular memory workload.

Data relocation \cite{14} is a software technique that dynamically reorganizes cache references in inner-loop array accesses. The reorganization yields a greater cache hit ratio and therefore reduces the number of accesses to memory. The reorganization is performed by mapping non-unit array accesses, as well as accesses from multiple arrays, into a sequential buffer in memory. First, a non-blocking \texttt{precollect} instruction loads cache lines with array elements which will be accessed, and the cache lines are tagged with the addresses of the allocated buffer. Subsequent memory instructions reference the buffer space instead of the original arrays. Then a closing non-blocking \texttt{distribute} instruction copies the cache data back to the original arrays. Synchronization instructions are also required to keep in-loop

\textsuperscript{4}This technique is vital for shared-memory multiprocessors, in order to prevent too much data sharing during execution.
loads from bypassing the non-blocking precollection instruction, which is loading from a different area of memory, and to keep post-loop loads from the original array from bypassing the non-blocking distribute instruction. The mapping of non-unit accesses into packed cache lines both increases spatial locality and decreases required space in the cache.

2.1.3 Hardware Solutions

While research in increasing memory bandwidth is continually ongoing, this section is devoted into looking at latency reduction techniques in the processor and/or cache. Cache organization techniques fall into this category [122, 62, 43, 119, 33, 18, 99, 64, 110, 127, 100, 57, 87, 77, 145], including such exotic schemes as trace caches [112, 115, 14]. Trace caches are instruction caches which are organized by dynamic trace instead of static program instruction order. The intention of trace caches is to keep the instruction window filed as long as fetch requests hit in the cache; trace caches do not themselves address the problem of memory latency due to cache misses.

2.2 Latency Masking Techniques

Because high memory latencies significantly slow down execution time even when locality and reuse in the cache are high, another category of techniques attempts to deal with cache miss delays by masking them with useful work in the processor.

The simplest of these techniques capitalizes on the ability to do this directly. For example, hardware-supported multithreaded architectures can perform explicit latency masking by switching contexts when the current thread encounters a long latency memory operation [68, 135, 3, 6, 95, 72]. Another thread can continue with non-memory operations while the memory request is being satisfied. Such architectures must address when to perform context switches [15, 75] and what program information constitutes the context [56]. Other architectures exploiting instruction level parallelism (ILP) and thread level parallelism (TLP) may perform implicit latency masking this way as well. For example, in a superscalar architecture, one instruction may be blocked on a memory access while other instructions requiring other machine units continue to execute independently.
These techniques work so well that the memory unit quickly becomes the critical resource, since memory operations are frequent even in optimized code. The solution is to combine these multiple-issue architectures with caches which support parallelism [70, 16, 96, 47, 89, 125, 41, 76]. These caches are typically called lookup-free or non-blocking caches, and they allow multiple simultaneous outstanding memory requests. This requires additional state indicating the current set of requests and where the results should be delivered, but since the memory operations are buffered, loads may bypass stores as long as the buffer is checked for overlapping addresses [52]. One study [126] showed that non-blocking loads coupled with an ideal load-completion policy could yield nearly ideal results. The authors used a simulator which allowed loads to be delayed indefinitely, and the order of completion was determined by the loads which, when delayed, caused the most performance degradation. This memory system yielded speedups within 8% of a perfect memory system (i.e., in which requests always hit in the L1 cache) on 8 SPEC95 benchmarks. This shows that the right combination of instruction scheduling and hardware speculation can yield nearly perfect results.

Even with a processing unit exploiting ILP or TLP and a non-blocking cache, however, the execution follows the pattern shown in Figure 1, where a significant amount of the processing unit’s execution time is spent waiting on memory operations. This can be alleviated through more latency masking. Like latency reduction techniques, latency masking techniques can be classified as compiler optimizations, software solutions, or hardware solutions.

2.2.1 Compiler Optimizations

Compilers can aid latency masking through the use of intelligent instruction scheduling [49, 69, 66, 1, 54, 97, 142, 146]. The ability of the compiler to schedule an overlapping of load latencies with other work is hampered by data dependences and dynamic control flow. If the system exploits a sufficient level of ILP/TLP, then good results are achieved as long as loads execute far enough ahead of their dependent instructions so that the data will be ready, or nearly so, by the time the dependent instructions execute. Due to limited compile-time
knowledge and limited available registers, it is not always possible to perform this type of code motion [4]. Additionally, loads cannot be moved to a place where they might generate a page fault that would not occur otherwise. For example, if a load instruction is added to an inner loop to get a value needed in the next iteration, it might cause a page fault in the last iteration by accessing outside an array, or even generate a page fault out of sequence. For the semantics of the program to remain the same, this is not allowed to happen.

2.2.2 Software Solutions

One way to ease aid code motion is to change the semantics of the load to avoid page faults. In one approach [111], special load instructions bypass the cache and use an interlock to ensure that any instructions which use the target register block until that value is available. These loads by design will not generate page faults; a special bit is set in the target register if a page fault would result, and this triggers the normal response if the register is later read. In another approach [29], special loads are allowed to proceed stores, even when the addresses may conflict at runtime. Special bits are used to indicate datum validity, and commit instructions are used to clear the bits and synchronize the target register with instructions executing in parallel. This approach can actually be implemented by the compiler alone, without the need for special instructions, using speculative execution [54].

These methods allow the compiler to place the loads in previous basic blocks or previous loop iterations, even if the address accessed is never needed in the execution. Both of these approaches are very much like software prefetching, discussed below, except that the semantics of software prefetching allows a simpler implementation and greater compile-time flexibility.

Many modern processors can execute instructions, even memory reference instructions, out of static program order. Typically, once the target address has been calculated, a load must wait for all previous unresolved stores, and a store must wait for all previous unresolved loads and stores, in order to make sure that memory references keep consistent with the original program order. Loads may bypass stores as soon as no conflict can be determined by the hardware. It is also possible, however, to allow loads to bypass stores
When data is accessed in a stream and therefore has little temporal locality, loading to and reading from a special FIFO queue [27] may yield results preferable to loading to and reading from a conventional cache. The compiler targets streaming loads inside loops and preloads the data needed for the next iteration into the queue. When the data is needed, it is retrieved from the queue and placed into a target register, just as in ordinary loads. In this case, however, the data which passes through the queue does not also pass through the cache and is not retained there. This technique requires exact compiler placement of instructions, since entries are inserted and retrieved in FIFO order, and the strict nature of the queue prohibits non-blocking access if the head of the queue is waiting to be serviced.

While software solutions typically suffer from a lack of runtime knowledge, this information can be estimated by profiling at compile-time, or obtained accurately by special instructions [63] which change control flow on a cache miss. These informing memory instructions allow the program to determine at runtime where misses are occurring and adaptively react. This is a large design subspace which depends on compiler technology to make use of the information.

2.2.3 Hardware Solutions

An independent hardware-based latency masking approach is through the use of speculation. Speculative execution performs an educated guess based on dynamic execution history as to which instructions will be required next. In speculative execution, the system initiates the memory request for these instructions (and continues into complete instruction processing) based on the outcome of the prediction; it does not wait for the next instruction address to be known for sure. In order to break up the long dependence chains due to loads, speculative hardware may predict either the load address or the datum returned.

In the first case, the load may proceed without having to wait for the load address to be calculated [9, 118, 51, 11]. A specialized approach called fast address calculation [9], works by predicting the address of an indirect load, where the address is calculated by adding an offset encoded into the instruction to a register operand containing a base address. When
the register operand is known, the load proceeds speculatively, predicting that the offset is zero. Help from the compiler can increase the frequency of zero-offset loads. When the load begins early in a speculative manner, the load latency is masked against the instructions which precede the load.

In the second case, the instructions following the load may proceed, using a predicted load result [82, 113, 46, 140, 20, 93, 94, 55, 130, 88, 80, 74]. In this way, the load latency is masked against the instructions which follow it, even if these instructions would normally be data-dependent on the load. Load value prediction can also be implemented as a software solution [140, 94, 74] in which the compiler determines when to speculate on the value and explicitly inserts instructions to recover from mispredictions. One study [74], however, found that while value prediction could be implemented completely in software, even a small hardware predictor table yielded much better results.

In the third case, loads are allowed to bypass stores speculatively at runtime, even if independence of the operations cannot yet be determined. If the bypassing load has read from a memory location to which the bypassed store has written, then the load and all its dependent instructions must be reexecuted.

These techniques have been successfully combined [118, 108] to mask memory latency. The physical cost incurred is additional hardware to recover from instructions executed due to mispredictions, and the performance cost is the time required to recover from mispredictions just as in branch prediction. A study by Reinman and Calder [108] which compared address prediction, value prediction, and dependence prediction, found that value prediction was the single most effective method, but that a combination of all three methods yielded the best results.

2.2.4 Combined Approaches

While the methods discussed above can be grouped into broad categories, many variations are possible. For example, a predominantly hardware-based scheme may be enabled and disabled by software instructions, or given prediction hints via software instructions. One approach [30] combined the hardware methods fast address calculation [9] and address
Figure 2: Increased memory latency masking through prefetching

prediction [82, 113, 86, 140, 20, 93, 94, 55, 130, 86, 80, 74] into a software approach in which loads were classified at compile time into normal loads, early calculation loads, and address-predicted loads. The combined approach allows strided accesses to be predicted and non-strided accesses to benefit from fast address calculation.

2.3 Prefetching

The dominant subcategory of latency masking techniques is prefetching. In prefetching, memory data and/or instructions are brought into the cache speculatively in the hopes that they will be needed in the future. In this way, the latency is incurred earlier, allowing more work to overlap and thus masking more or all of the latency (see Figure 2). There is the problem that the target address of a prefetch might generate a page fault, but non-binding prefetches have the advantage that they may be discarded completely in such a case. Prefetching research overlaps the latency masking technique categories of compiler optimizations, software solutions [104, 50, 21, 67, 89, 99, 1, 143, 144, 81, 23, 138, 139, 84, 116, 85], and hardware solutions [122, 123, 78, 63, 44, 10, 45, 24, 121, 124, 83, 38, 98, 26, 5, 103, 102, 61, 12, 114, 71, 109, 32, 58].
2.3.1 Speculation Techniques

Since prefetching predicts the address of a datum which will be required in the near future, the prefetching components must make one or more assumptions about the underlying access patterns. These fall into general categories.

2.3.1.1 Temporal Locality

One common form is temporal locality, the tendency for memory data to be accessed repeatedly. It is this locality which is the basis for using a cache system.

2.3.1.2 Spatial Locality

Another common form is spatial locality, the tendency of future accesses to have memory addresses near to recent accesses. The idea of grouping cache words into lines is based upon this idea; if the first word of a line is accessed, it is likely that the remaining words will be as well. Like temporal locality, this tends to be true for instructions as well as data.

2.3.1.3 Stride-based Address Prediction

An extension of spatial locality is the idea of stride-based address prediction for loads. Regular data accesses produce distinct stride patterns. Take as an example a nest of $n$ loops which are used to access a D-dimensional array, $A$:

```c
for(i1=0;i1<t1;i1++) {
    for(i2=0;i2<t2;i2++) {
        for(i3=0;i3<t3;i3++) {
            (etc.)
            f(A[a1][a2][a3]...);
        }
    }
}
```

The array will be mapped to memory by scaling the array indexes, as follows:

$$\mathcal{A}[a_1][a_2][a_3]...[a_D] = A_{base} + a_1 s_1 + a_2 s_2 + a_3 s_3 + ... + a_D s_D$$  \hspace{1cm} (2)

16
Or, more succinctly:

\[ \&A[a_1][a_2][a_3] \ldots [a_L] = A_{base} + \sum_{j=1}^{D} a_j e_j \]  
(3)

If the array indexes \(a_1, a_2, \ldots, a_D\) are linear combinations of \(\{1, i_1, i_2, \ldots, i_n\}\) as in:

\[ a_j = c_{j,0} + c_{j,1} i_1 + c_{j,2} i_2 + \cdots + c_{j,n} i_n, \]  
(4)

Then from Equations 3 and 4:

\[ \&A[a_1][a_2][a_3] \ldots [a_D] = A_{base} + \sum_{j=1}^{D} s_j (c_{j,0} + \sum_{k=1}^{n} c_{j,k} i_k) \]  
(5)

\[ = A_{base} + \sum_{j=1}^{D} s_j c_{j,0} + \sum_{j=1}^{D} \sum_{k=1}^{n} s_j c_{j,k} i_k \]

\[ = A_{base} + C_0 + \sum_{k=1}^{n} \sum_{j=1}^{D} s_j c_{j,k} i_k \]

\[ = A_{base} + C_0 + \sum_{k=1}^{n} i_k \sum_{j=1}^{D} s_j c_{j,k} \]

\[ = A_{base} + C_0 + \sum_{k=1}^{n} i_k C_k \]  
(6)

where \(C_0 = \sum_{j=1}^{D} s_j c_{j,0}\) and \(\{C_k = \sum_{j=1}^{D} s_j c_{j,k}\}\) represent new constants.

As the program walks through iterations of the loop nest, the set of iteration variables will change from \(\{i_1, i_2, \ldots, i_n\}\) to \(\{i_1', i_2', \ldots, i_n'\}\), causing the address accessed to change from \(A_{base} + C_0 + \sum_{k=1}^{n} i_k C_k\) to \(A_{base} + C_0 + \sum_{k=1}^{n} i_k' C_k\), a difference of \(\sum_{k=1}^{n} (i_k' - i_k) C_k\).

Usually, \(i_n'\) will be \(i_n + 1\), and the rest of \(\{i_k'\}\) will be \(\{i_k\}\). In this case, the difference will be simply \(C_n\). When \(i_n\) reaches \(i_n\), it will revert to \(0\), and usually \(i_{n-1}\) will increase by one. This difference, then, is \(C_{n-1} - (i_n - 1)C_n\). Less often, both of these indexes will roll over at the same time, and the difference will be \(C_{n-2} - (i_{n-1} - 1)C_{n-1} - (i_n - 1)C_n\), etc.

In short, the stride pattern will be \(C_n\), interrupted every \(i_n\) by another value, one which varies.

2.3.1.4 Control-flow Target Prediction

Instruction prefetching depends on prediction of control flow. Instructions normally imply incrementation of the program counter, but instructions which can change control flow set the program counter to a different value. Spatial locality holds for sequential instructions.
in a basic block, but not for instructions which change control flow. Control-flow target prediction refers to methods which predict the target instruction from a control flow operation. This does not refer to whether control flow is changed, but rather how. Often, but not always, the target is statically encoded into the instruction.

2.3.2 Problems with Prefetching

Ideally, prefetching merely initiates necessary memory accesses early so that more of the incurred latency can be overlapped with instruction execution. In practice, however, two independent problems may arise. The first of these is misprediction. When misprediction occurs, an unneeded cache line is requested from memory, increasing bus utilization. This increased bus usage may delay needed memory requests, thus having the opposite of the desired effect.

The other problem that may occur is increased cache contention. A line \( L_p \) is prefetched, and it displaces a line in the cache \( L_q \). If \( L_q \) is needed before \( L_p \), then \( L_q \) will have to be retrieved from memory, displacing \( L_p \). In this case, the prefetch request caused an additional access to memory, increasing bus utilization and delaying the instructions which depend on the contents of \( L_q \).

2.3.3 Variants of Prefetching

Prefetching techniques can be categorized by five axes: the hardware component, the software component, the compiler support, the prediction method, and the trigger. These techniques tend to fall into only three general categories and combinations thereof, however.

2.3.3.1 Software Prefetching

Software prefetching is a category of prefetching techniques in which the trigger is the execution of a special prefetching instruction. These approaches rely on compiler technology to deduce memory access patterns at compile-time. Based on these access patterns and placement of memory-requesting instructions, the compiler inserts the prefetch instructions into the executable. The prefetch instructions ensure that the cache line(s) which they specify
are present in the cache; if they are not present, a request for them is issued to the next level of the memory hierarchy. In this way, by the time that the normal memory-requesting instructions execute, the data will be in the cache. It is only through the execution of the prefetch instructions that such prefetching occurs.

The software component of these approaches is the added set of prefetching instructions, which generally are unconnected to the data flow of the program. The hardware component is the extra functional unit required to execute the prefetching instructions.

Software prefetching methods differ by the compiler algorithm which constructs and places the instructions and by the type of instructions available. These instructions initiate memory operations, but the results of these operations affect the state of only the memory subsystem. The object of the operations is to incur the memory operation latencies early.

Software prefetching has the advantage of knowing, at any given instruction, that instruction’s context in the overall program: the frequency of memory operations, the paths that may be followed (and therefore the possible upcoming loads), and the regular access patterns of arrays.

Hardware support for these instructions does exist in architectures such as the DEC Alpha [39] and HP PA-8000 [114].

2.5.3.2 Compiler Optimizations

In addition to code motion and instruction scheduling, which do not qualify as prefetching techniques because they are not speculative, there is at least one form of prefetching which can be accomplished without any specialized hardware or software. This is a variant of software prefetching in which regular loads are placed instead of prefetch instructions. In order to allow these instructions to execute and not affect the program’s data flow, each prefetching load’s destination register must not be live at the placement of the new instruction. This requires a modification to the register allocation algorithm of the compiler. The easiest approach would be to reserve one integer register and one floating-point register as destination registers for prefetching loads. Then register allocation could proceed using the remaining registers.
Such an approach requires no hardware components. The software component is the set of added load instructions (instead of special prefetch instructions), and the compiler support is the same as for software prefetching except for the change in register allocation. Likewise, the trigger is the execution of the additional loads instead of execution of prefetch instructions.

2.3.3.3 Hardware Prefetching

Hardware prefetching is a category of prefetching techniques in which the trigger is a hardware event and the prediction is performed in hardware at run-time, not at compile-time. There is no software component and no compiler support in hardware prefetching.

Hardware prefetching has the advantage of run-time information which can only be predicted at compile-time, such as the exact sequence of memory operations, the instructions which initiated them, and when they occurred, as well as a record of cache behavior. In addition, hardware prefetching methods do not increase the workload of the CPU with the introduction of additional instructions to execute, nor do they increase the workload of the I-cache system by increasing the code size.

Hardware prefetching methods differ by the hardware components, the run-time information they track, the triggers they use, and the predictions they make. There are broad categories of hardware prefetching methods, however.

2.3.4 Prefetching for Shared-Memory Architectures

In shared memory machines where each processor retains a cache of the shared memory, special techniques are required to ensure consistency between the caches [8]. This complicates the semantics of prefetching slightly. Binding techniques [79, 50] bind the location of the data block to the value kept in the cache so that consistency can be checked when the data is actually accessed. Non-binding techniques [89, 128] resolve consistency at the time the prefetch is issued by changing the ownership of the block or whatever the mechanism requires.

Any hardware technique employing buffers complicates cache consistency on these types of shared memory machines. There is the question of how/when to maintain consistency.
of buffered values, as well as the fact that writes may not be seen immediately by the other processors. Generally, the programmer's memory consistency model is relaxed to accommodate the possible effects of many processors referencing the same shared memory. The most strict usable consistency model is called sequential consistency, which means that the effects of the memory operations on multiple processors could be achieved by some sequential interleaving of operations on a single processor. Many models which relax this requirement have been proposed [36, 2, 47, 37, 147]. In order to get sequential consistency in program critical sections, it is necessary for the compiler, via explicit statements or hints from the programmer, to insert special synchronization mechanisms into the program.

Another vital issue in shared memory architectures is the amount of memory traffic which any prefetching method generates. Tullsen and Eggers advised against software prefetching altogether for shared-bus architectures [129], and Ranganathan et al. [106] reported disappointing results on even two-dimensional mesh architectures.

2.4 Related Work

This section presents a survey of previous work in the area of speculation-based latency masking. First, software approaches\(^2\) are discussed, and then hardware approaches\(^3\).

2.4.1 Software Approaches

2.4.1.1 Regular Software Prefetching

Porterfield et al. [104, 21] studied the approach of prefetching array accesses in a loop one iteration ahead. Simulations of scientific programs revealed that this approach on such programs was more effective than such hardware approaches as long cache lines and one block look-ahead (OHL). Execution times were decreased 20% on average, and up to 50%. Experimental results demonstrated that this approach generated too much overhead, and so a selective approach, where prefetching was performed only for memory accesses likely to generate misses, was investigated. The selective approach eliminated 54% of the prefetch instructions from the benchmarks selected. Additionally, they investigated the possibility

\(^2\)as defined above: compiler technology supported by hardware 
\(^3\)referring to hardware technology to improve executables without recompilation
of storing prefetching addresses in registers to be used as the actual load addresses during the next iteration. This would prevent these addresses from being calculated twice unnecessarily. The preliminary results were inconclusive as to whether this compiler optimization would speedup benchmark execution.

Mowry et al. [99] addressed the overhead of adding prefetch instructions. They developed a compiler algorithm which used locality analysis to determine which memory accesses were most likely to benefit from added prefetch instructions. Their study was limited to dense matrix code, however.

2.4.1.2 Software Prefetching in Shared Memory Architectures

Gornish et al. [50] proposed an algorithm for parallel programs on multiprocessor architectures. The algorithm prefetches a block of an array used in a parallel loop as early as possible, in order to minimize interprocessor communication in execution of the loop. Binding prefetching is used so that the approach is independent of the underlying cache coherence mechanisms. Simulation of executing Fortran code showed a reduction in processor stall time from 32% to 97%.

Mowry and Gupta [89] proposed a non-binding prefetching mechanism; in it, prefetches are treated semantically as early loads, and the hardware cache coherence mechanism enforces consistency. Sacrificing the generality of [50], the non-binding approach may capitalize on additional prefetching opportunities, being free from control flow and data dependency constraints which binding prefetching introduces. For simulations of programs with hand-inserted prefetch instructions, performance of regular programs was as high as 86%, and up to 14% for an irregular program dominated by accesses in a linked list. Their study included prefetch instructions for writes as well as reads, and for variable-sized prefetching.

2.4.1.3 Scheduling Based on Profiling

Abraham et al. [1] used a combination of compile-time prediction and program profiling in order to predict which memory operations were likely to be the few which caused the majority of the program misses. This information was used to perform selective scheduling, which assigned longer latencies to memory operations which were likely to trigger a cache
miss. They also introduced two two-bit modifiers to load instructions which are placed by the compiler, and which control what levels of the cache hierarchy may retain the load target data. Traditional caches retain data at all levels, but the modifiers allow the cache to specify that a load result not be retained at any level of the cache hierarchy, which it would choose to do if the reuse for that load were very low. Abraham et al. provided detailed mathematical analysis as a basis for a compiler algorithm, but this was done for only one simple example. The issue of a general algorithm that includes irregular code was left open.

2.4.1.4 Software Instruction Prefetching

While software prefetching is almost always a form of data prefetching, a form of instruction prefetching in software was proposed by Young et al. in [143]. In this method, instructions are inserted before branches to advise about prefetching from the branch target. The limitation of this approach is the same as that of software data prefetching: as memory latencies increase, the instructions must be placed further and further ahead of the target loads/fetches. This means moving the instructions into previous basic blocks and therefore sacrificing the accuracy of the prediction (i.e., whether the request will be used or not).

A similar approach was taken by Xia and Torrellas [139], in a technique called guarded sequential prefetching. This is sequential prefetching that starts at the program counter and continues linearly through the static layout of the code until a control flow instruction is predecoded which sets a guard bit. This approach did not attempt to prefetch from branch targets when the targets did not follow linearly from the current basic block; instead, it relied on a sophisticated algorithm for layout code in memory to maximize spatial locality by assembling compile-time-predicted traces. This technique was tested on operating system code, and it achieved a 10% speedup.

Luk and Nowry [85] expanded on ideas by Young et al. [143] and Xia and Torrellas [139] by performing sequential prefetching in hardware on the L1-cache in addition to the execution of instructions which triggered prefetching of the most likely upcoming control flow target. This method did not rely on predecoding of control flow instructions to halt the sequential prefetching. Instead it used a new type of safety valve; each L2 cache line
was augmented with saturating two-bit counters, like the kind used in branch prediction. The counters were incremented when a prefetched L2 line was referenced by the I-cache, and decremented when a line prefetched from the L2 into the I-cache was displaced without being referenced by the I-cache. Then prefetch requests for L2 lines with low counter values were ignored. This method yielded an average speedup of 13.3% on seven selected integer benchmarks, masking 71% of fetch latency. This approach suffers from the same problems as that of Young et al. [143], in that the prefetching instructions have to be placed far enough ahead in the instruction stream to be useful, and that the addition of instruction prefetching instructions hurts the available I-cache bandwidth while attempting to exploit it better.

2.4.1.5 Additional Specialized Hardware

Klaber and Levy [67] suggested prefetching data into a separate buffer, in an algorithm which uses memory latency and loop execution time to place the prefetch instructions. Chen et al. [28] showed that such a prefetch buffer is more effective than a corresponding increase in cache size, for small caches.

Xia and Torrellas [138] studied data cache operation during the execution of operating system code. They advocated DMA-like pipelining for block operations in the majority of cases, with the remaining cache hot spots handled by software prefetching. This approach yielded a 19% speedup in the operating system code, with 75% of miss latency masked.

Chen [23] studied the idea of using a prefetch engine. This is a unit which is responsible for keeping track of instructions for which the compiler wants prefetching to occur. The unit receives its instructions when special instructions are executed, but one instruction placed before a loop can take the place of an instruction inside the loop which would be executed every iteration. The compiler tells the engine at what PC to prefetch, the base address, the stride, how often to prefetch, and when to begin. The chosen PC is not necessarily that of the load address; the compiler determines an address sufficiently far ahead of the load to mask the miss latency. The address might even appear after the load address in the loop, if the prefetching is for the next iteration. When the PC of an entry in the engine's
internal table is encountered, the entry's internal counter is incremented, wrapping around based on the frequency. Only when the counter is reset to zero does prefetching occur. Then the entry's address is incremented by the stride. This allows the prefetch engine to not only prefetch every \( x \) bytes, but also do so only every \( y \) iterations. Thus, prefetches can be avoided when the compiler determines, for example, that every other access will hit in the cache. This is much preferable to traditional approaches, which, either avoided the unnecessary cache access by unrolling the loop by a factor of \( y \) and inserting prefetches in only one of the unrolled loop bodies, or by testing a modulo division of the loop counter by \( y \) and using the result to branch around the prefetch instructions. Unrolling loops can be unwise if the unrolled size is greater than that of the I-cache, and inserting the modulo test branch in a loop would cause a misprediction every \( y \) iterations for most branch predictors, and this would flush the pipeline and therefore slow down execution. For a 32K D-cache with a line size of 32 bytes, Chen achieved lost cycle recovery rates from practically 6% to practically 100%. The instruction placement algorithm was able to cheat a bit, however; a dynamic instruction stream was analyzed instead of static code. This would enable the algorithm to place instructions even when it would be nearly impossible to do so at compile time.

\[\text{2.4.1.6 Non-array Based Software Prefetching}\]

Zhang and Torrellas [144] studied the problem of software data prefetching for irregular code. Their method relies on code which makes use of abstract data type(s) (or objects). The compiler determines which fields are often accessed together; then, pending approval from the programmer, the compiler records data type fields so that related fields are stored together spatially in memory. These groups of fields are represented as bound in memory through the use of extra bits. The programmer supplies hints as to which groups are logically related, and the compiler links related groups through the extra hardware bits. Prefetching is done not through instruction execution but by hardware which recognizes group access and issues prefetch requests for unavailable fields and for related groups. It prefetches only one link level at a time, which the authors dub \textit{pipelined prefetching of groups}. This method
achieved speedup for 6 of 7 benchmarks, up to 25 to 40%.

Lipasti et al. [81] investigated non-scientific code which tends to be littered with pointer dereferences and procedure calls. They tried a simple heuristic for inserting prefetch instructions for such programs: they prefetch the target cache line of pointers passed as arguments in procedure calls. This technique reduced cache misses by 5-7% on average, depending on cache size, up to as much as 20% for one benchmark.

Luk and Mowry [84] studied prefetching for regular code but for non-array data structures, specifically recursive data structures such as linked lists and binary search trees. The problem with such data structures is that the needed address for the next loop iteration cannot always be determined absolutely; it may be contained in the data needed in the previous iteration. In addition, it may be necessary to prefetch multiple iterations ahead in order to fully mask the memory latency, and this can require multiple accesses to memory in order to obtain the needed address. Luk and Mowry investigated ways of performing the prefetching when the needed address was one memory access away and when the needed address was available in the current iteration’s data structure (i.e., zero memory accesses away). They looked at ways to prefetch for algorithms which met this criterion, and they added address pointers into the data structures in order to achieve this criterion in other algorithms. These added pointers, called history pointers, are set during the first traversal of the data structure; subsequent like traversals would use the history pointers as prefetch addresses, while a different type of traversal would not be able to use the values of the history pointers. In such a case, multiple values of the history pointers can be maintained for each type of traversal performed. Results showed a speedup from 4% to 45% across ten benchmarks for machine-compiled code, and much better results for hand-compiled code.

Roth and Sohi [115] combined this type of prefetching with traditional chained prefetching, and achieved speedups of 15% on average with a software-based approach. Interestingly, a hardware implementation yielded speedups of 22% on average, although a cooperative approach worked best when the benchmark made only a single traversal through the data structures.

Karlsson et al. [65] built upon this work, expanding the use of a single history pointer
to an array of history pointers, each of which points to a possible accessed node. In the case of a binary search tree, in order to prefetch two iterations ahead, an array of four history pointers is needed to cover each of the four nodes which may be accessed by accessing two levels deeper into the tree. They classified six pointer-intensive kernels as linked list or tree-based. For the linked list kernels, they masked 23% to 51% of memory latency, and for the tree-based kernels, they masked 60% of memory latency.

2.4.1.7 Software Prefetching with Runtime Feedback

Mowry and Luk [91, 92] investigated using informing memory instructions [53] to adapt to dynamic conditions, in a technique they called correlation profiling. With this technique, feedback from memory operations is used to correlate load instruction results with control flow (i.e., the most recent $N$ basic blocks), previous self results (i.e., the most recent $N$ hit-or-miss results from the same load), or previous global results (i.e., the most recent $N$ hit-or-miss results from any and all loads). The prediction outcome can then be used to determine whether a prefetch instruction should be executed. Results from using this method were very modest, however; three benchmarks with hand-inserted prefetch instructions yielded only a 3% speedup using this method.

2.4.1.8 Real World Software Prefetching

Unlike the studies above, Santhanam et al.[116] studied software prefetching on a real architecture, the HP PA-8000. Their instruction insertion algorithm reduces each array index expression to a linear function of the loop counter. Then the sizes of strides and the inter-spatial locality of instructions in the same loop are considered so that no more prefetch instructions are inserted than are necessary to insure coverage of the loop's accesses. Their results were very good, up to nearly 100% speedup and a geometric mean of about 40%, but the analysis was performed for the SPEC95fp benchmarks, which are very regular and predictable at compile-time.
2.4.1.9 Software Prefetching Summary

All software approaches suffer from the cost incurred by adding prefetch instructions, from a limited amount of data access patterns which can be determined statically (i.e., at compile time), from determining when prefetching should take place, and from placing the instructions into the resulting code. For software approaches which attempt to adapt to dynamic conditions at runtime [91], this involves trading limited compile-time information for increased overhead in the form of added instructions. A 1997 study comparing hardware and software approaches on HP architectures revealed the superiority of a simple hardware approach over software prefetching [132]. In 2001, another hardware approach was shown to reap almost all the benefit available from software prefetching [40]. Research issues for software approaches continue to be when and where to insert prefetch instructions, flavors of prefetch instructions, and novel ways of reducing the inherent overhead of executing the prefetch instructions.

2.4.2 Hardware Approaches

2.4.2.1 Cache Organization

One way to perform a form of automatic prefetching is through the use of long cache lines. The cache is organized with long cache lines in an attempt to exploit spatial locality. There is no additional hardware component required; the trigger is handled by the cache system itself. As with shorter cache lines, as soon as one word is requested, the other words in the line are also requested. In the case where a cache line is too large to be transmitted all at once, the cache logic is usually designed to give priority to the piece of the cache line containing the request. One problem with long cache lines is that, while memory accesses tend to progress forwards, a request for the last cache word in an long cache line will trigger the fetching of all previous words in the same line instead of the words in the next line. Another problem is that as cache lines become larger than the average spatial locality, words are fetched which are never used, wasting bus bandwidth.

Logriore [83] proposed reserving cache lines specifically for stack data words. Some optimizations would then be possible, as normal program semantics indicate that when the
stack shrinks, data previously on the stack are invalid and do not need to be written back to memory. The high locality of stack accesses allow stack data to be prefetched with a high accuracy. The limitation of this approach is that it is limited to stack accesses. Many applications use either large global arrays or dynamically allocated data, and no prefetching is done for these data.

2.4.2.2 Forms of One-Block Lookahead

Smith [122] studied variations on one block lookahead (OBL) prefetching schemes. In OBL schemes, shorter cache lines are used, but the request of cache line \( i \) will also initiate prefetching for cache line \( i + 1 \). Variations on this approach include changing the number of blocks to look ahead and allowing negative lookahead (e.g., requesting cache block \( i - 1 \)), based on detected access patterns. Smith showed results which would characterize prefetching research: an improvement in cache hit ratio, balanced by an increase in memory traffic caused by unused prefetches.

Jouppi [63] extended the OBL approach to parallel FIFO queues called stream buffers. Each stream buffer is independent of the others and represents a series of cache blocks (i.e., lines). Memory requests that miss in the primary cache(s) are checked against the stream buffers, and if the block is not found in the stream buffers, then a new stream buffer is allocated for that block. Stream buffers are serviced by the next level of the memory hierarchy in a round-robin fashion, each requesting the next adjacent block until it is full. When a cache block in a stream buffer fulfills a cache miss, the stream buffer shifts to pass out the cache block, creating one or more empty entries on the end. Additionally, in order to alleviate conflict misses, Jouppi introduced the concept of a victim cache. This is a small but fully-associative cache which stores recently displaced cache lines. Misses in the primary cache are checked against the victim cache, and in case of a hit, the cache line moves from the victim cache to the primary cache. Thus, the contents of the primary cache and of the victim cache are mutually exclusive. Jouppi’s experiments showed that a stream buffer of 4 cache lines, at 8 bytes per line, reduced misses on a 4K I-cache by 85% and by 35% on a 4K D-cache. This difference in effectiveness is due to the higher degree of spatial
locality of L-cache accesses. Data accesses with a large regular stride would require a much
greater stream buffer to be effective, and even then much unnecessary data would be fetched
between the requested data.

Palscharla and Kessler [88] investigated using stream buffers instead of an L2 cache to
support the L1 cache. The stream buffer design was enhanced to determine non-unit strides,
using the method described in Section 2.3.1.3, with a history buffer of independent access
patterns. They found that results using the stream buffers were competitive with using a
large L2 cache, while requiring less die area. They concluded that stream buffers could be
employed in design as a way of reducing the cost of the machine.

Farkas et al. [42] looked at the interaction of stream buffers with non-blocking loads
in the SPEC92 benchmarks. They found that non-blocking loads alone yielded a 21% reduction
in execution time, and stream buffers alone yielded a 26% reduction in execution
time. Together, the benefits were cumulative, a 47% reduction in execution time.

Bennett and Flynn [12] enhanced the victim cache and stream buffers with a cache miss
history buffer. This is a FIFO queue which stores the last ten cache miss indexes (i.e., the
direct-mapped segment of the target address). When a miss in the cache matches a recent
sequence of cache miss indexes, this indicates a cache hot spot, and the displaced address is moved into
the victim cache. If a miss in the cache is adjacent by one line to an entry in the history
buffer, this indicates either a forward or backward stream, and prefetching is performed
for n lines ahead (or behind) in the stream. The value n starts off at one and can double
or halve based on the frequency of partial hits (i.e., cache hits where the cache line has
been requested but has not yet arrived). This approach yielded a higher ratio of cycles
recovered to cache misses, while simultaneously requiring less bandwidth than traditional
stream buffers. It did not, however, address the issue of streams with a stride of two or
more cache lines. These would not manifest themselves as streams and would therefore not
trigger any prefetching.

Johnson et al. [59] used a spatial locality detection table which was actually just an
extension of OBF, which allowed the hardware to determine at runtime when adjacent
cache lines should be prefetched. Their reasoning was that not all adjacent cache lines

30
exhibit spatial locality, so a detection mechanism would reduce unused prefetched cache lines. They achieved results ranging from a very slight slowdown to about a 9% speedup across ten benchmarks from the SPEC92 and SPEC95 suites.

Kumar and Willerson [71] looked back to the OBL-based approaches to define spatial footprints. In this method, the memory address space is sectored, and the knes requested within a sector are stored in a bit vector until a cache request lies outside the sector, at which time the bit vector is kept in a history table with the first instruction accessing the sector and the first data address accessed in the sector are stored. When an inactive sector is referenced, the same footprint (i.e., pattern of accessed lines within the sector) is prefetched. The footprint may be shifted from the original if the first data address accessed in the sector is not the same as the first data address as stored in the history table; this allows the mechanism to immediately adapt to migration of the spatial footprints caused by array accesses, and this is the primary difference between this work and that of Johnson et al. [59] above. Of the predictors that Kumar and Willerson simulated, the one yielding the best benefit/cost ratio was able to reduce the miss ratio by 18% on average. This predictor stored just the first data address accessed per sector and one level of history, which required 8.75 Kb of on-chip storage. This is a considerable amount, given that the cache simulations were based on a 16 Kb data cache.

Lin et al. [40] used a similar approach on L2 caches, using the term region for the cache sectors. Unlike spatial footprints, the regions are simple sectors and not subsets; the entire region is always prefetched. Only unused cycles were used, and requests were services in LIFO order. Their research was on Direct Rambus DRAM architectures, and so they were able to introduce some architecture-specific optimizations to reduce bank conflicts as well. Finally, as the prefetching was directed to the 4-way set associative L2 level, they were able to address cache pollution by assigning the prefetched blocks to the lowest priority in their respective cache sets. This made them first-eligible for displacement on a conflict. The optimized hardware configuration improved performance by 16% on average, and prefetching improved performance by an additional 43% for 10 of the 26 benchmarks studied.
Alexander and Kedem [9] also used a cache sectoring approach, although in a very different way. Their idea was to divide memory into equal blocks of 128 to 512 bytes. Then the sequence of block references is monitored, and pairs of adjacent references are stored. If a reference in block a was recently followed by a reference to block b and the current reference is to block a, then block b will be prefetched. Their analysis showed that this prediction technique would yield an accuracy of up to 97%, although in practice one would expect this to result in lots of wasted bandwidth due to unused words within blocks. This is the same argument against using very large cache lines. Additionally, they advocated using a prediction table of 1% of the total memory size. For just 64 megabytes of main memory, this would be a 640 kilobyte table, an order of magnitude larger than the primary cache.

Collins and Tullsen [32] used a miss classification table to determine whether cache misses were capacity/compulsory or conflict misses. They developed a strategy in which evicted lines in conflict misses were sent to a victim cache, and next-line prefetching was performed for capacity misses. This technique yielded a speedup of 16% over either approach alone.

2.4.2.3 Stride-based Prefetching

Fu and Patel [44] proposed a prefetching strategy specifically for vector processors. They defined the cache load size \( l \) as the number of bytes requested from the next memory level on a cache miss. They gave this value as \( l = (p+1) \times b \), where \( p \) is the number of prefetched cache blocks\(^4\), and \( b \) is the size of one cache block in bytes. They then categorized sequential prefetch strategies as those which prefetch \( p \) consecutive cache lines on a miss when the stride is less than \( b^2 \). They defined stride prefetching as a strategy which employs sequential prefetching and also prefetches \( p \) non-consecutive blocks for accesses with strides greater than or equal to \( b \); the blocks accessed are separated by the stride length. Simulations of four numerical codes on a 2-way 64K cache using \( p = 3 \) and \( b = 32 \) showed a 30-50% speedup for sequential prefetching, and only slightly better for stride prefetching.

\(^4\)C.e., lines
\(^5\)including scalar accesses, which have a stride of 0
Dahlgren and Stenström [35] compared stride-based prefetching and sequential prefetching for six selected benchmarks. Their results were mixed; they found that while stride accesses dominated the runtime characteristics of four of the six benchmarks, sequential prefetching yielded greater performance in three of the six. They found that this was because the strides themselves tended to be small, and so several accesses would fit into a single cache line. Stride-based prefetching, however, tended to be less taxing of memory bandwidth.

2.4.2.4 Markov Predictor

Joseph and Grunwald [61] used memory trace simulations to show that a Markov predictor outperformed both stream and stride based prefetching for traces of multiple processes including kernel, user, and shared library activities. Their simulations showed on average a 54% reduction of execution stalls when the Markov predictor was employed. This predictor, however, has a high hardware overhead (over one megabyte for prefetcher data structures), and is intended as an interface between an on-chip cache and an off-chip cache or memory.

Sherwood et al. [20] used stride filtered Markov predictors as the prediction mechanism in a stream buffer implementation. They added confidence predictors so that high-confidence sequences could be given a higher priority, and so that sufficiently low-confidence sequences could be dropped entirely. On six selected benchmarks, this approach yielded a 30% average speedup, which was a 10% improvement over traditional stream buffers.

2.4.2.5 Lookahead Prefetching

A subcategory of hardware prefetching approaches is lookahead prefetching. These approaches attempt to predict upcoming loads by an early or lookahead decoding of the instruction stream, or by accessing a buffer of previously decoded instructions. A predecoded load instruction serves as the trigger for a prefetch operation. Decoupled architectures [123] may do this implicitly; one instruction stream feeding the data access unit may stay ahead of the other instruction stream feeding the functional units. If so, then the memory operands fetched by the former instruction stream can be available to the latter instruction.

33
stream when needed. To be effective, this approach requires a CISC-like instruction set, so that the data access stream rarely needs information from the functional unit stream. Jones and Toppan [90] showed that decoupled architectures are better able to prefetch than superscalar machines, for smaller instruction windows. Lee et al. [78] proposed a lookahead scheme for single stream CISC architectures. A large instruction prefetch buffer is used to store instructions, and a data prefetch buffer is used to store addresses for decoded loads. As soon as the address is made available to the data prefetch buffer, a prefetch is issued for the address. If instructions are decoded far enough ahead of execution, then the prefetch may have time to supply the cache with the needed datum. On conditional branches, the buffering continues down a random path; when such predictions are proven incorrect by branch execution, the systems stalls while the instruction prefetch buffer is flushed and corrected.

Baer and Chen [10, 24, 26] developed a data prefetching scheme to accommodate variable-length data access strides. They keep a reference prediction table (RPT), which is associative by load instruction address. The most recent load, most recent matching stride, and a two-bit state are kept for each entry. Dedicated hardware attempts to keep a lookahead program counter (LA-PC) a fixed distance d ahead of the machine PC by using a branch prediction table. This table stores branch instruction addresses and their associated predicted targets and is filled by the normal decode and branch prediction units. The lookahead unit reads this table to follow the route which the PC should follow. When an upcoming load is encountered (via a hit of the LA-PC in the RPT), the RPT predicts the load target address using the available state for that entry and issues a prefetch request from the D-cache. They recovered from 10% to 95% of cycles lost to misses from a 32K D-cache across four benchmarks. Their study was limited to trace-driven simulations of single scalar architectures assuming perfect L-cache and next-level behavior, as well as perfect RISC pipelining. They did not simulate a full machine to determine how well the LA-PC would be able to stay ahead of the PC with an imperfect branch predictor, nor how hit-wait times introduced by a non-blocking cache would affect the ideal value for d. Additionally,

This information is passed in queues which serve as the interstream communication mechanism
the RPT does not scale well to superscalar architectures, as predictions are made by an associative LA-PC lookup each cycle; if the LA-PC can change by 4 or larger, the RPT would need to perform potentially as many lookups each cycle.

Following up on their spatial locality prefetching research for vector processors, Fu and Patel developed a lookahead prefetching technique for scalar processors [45]. Similar to Baer and Chen's method [10], they keep a history table of the most recent stride per load instruction. Their results showed that even a small history table can greatly reduce the cache miss ratio on vectorizable programs. Their scheme was unable to detect irregular access patterns, however, and therefore programs laden with irregular accesses inadvertently generate many spurious prefetches, burdening the memory system. Sklenar [121] proposed a similar method, although without experimental results.

Eckemeyer and Vassiliadis [38] proposed a lookahead scheme in which a prefetched data value is bound to the program data flow. In their scheme, data prefetches are triggered by loads decoded in the instruction buffer, and the value prefetched is made available to the later computations which use it. Addresses are predicted via a stride predictor similar to Baer and Chen's method [10], and the instruction buffer is large enough to hopefully decode the load in time to provide the loaded datum without a delay. This is essentially the principle behind branch prediction applied to loads; the result from the execute phase is predicted early, and if this value turns out to be incorrect, the pipeline must be stalled and computation resumed from the point at which the correct execute value is known. In this case, the value calculated in the execute phase is the load address. This scheme depends on the length of the pipeline and instruction buffer to mask the memory latency for cache, and incorrect address prediction will stall the pipeline and cause a flush even if the value from the correct address is in the D-cache. The more traditional methods above have the advantage that when the predicted address is wrong, only bandwidth has been wasted; the prefetching method supplies data to the cache, but not to the processor. In addition, predictions made from regular accesses will greatly hamper their ability to provide the correct data to the pipeline.

Pinter and Yuan [102] also built directly upon Baer and Chen's method [10, 24, 26]
by building a Program Progress Graph (PPG) in hardware, which allows the prefetching mechanism, via shared branch prediction, to traverse the path of execution one basic block per cycle. The PPG is expanded to include PPG tags, and is called the SRPT. Data prediction occurs by associatively scanning the SRPT for the current PPG tag. Prediction requests are queued into a Prefetch Requests Controller (PRC). While queued, the requests are filtered once against duplicates already queued as well as recent requests which hit in the data cache upon release from the queue. The requests are only released when there is an unused data cache port. Their trace-driven simulations of SPEC92 benchmarks showed speedups of 2% to 83%. Unfortunately, the PPG scheme is not scalable with wider-issue architectures. Since the average size of a basic block is only 4.5 instructions for non-scientific code, this severely limits the ability of their pre-PC to get ahead of the PC. And for scientific code, the average distance between memory operations is less than 8 instructions. Since their scheme can handle at most two memory operations in one associative pass of the SRPT, this also limits the ability of the pre-PC to get ahead of the PC, albeit not as severely. Additionally, since prefetch requests are queued and there is a delay in getting them dispatched, anything limiting the ability of the pre-PC to get ahead of the PC is a hindrance as memory latencies increase. Despite these problems, however, the PRC seems to be an ingenious way of enabling prefetching without increasing the number of ports on the cache.

Annarum et al. [7] combined lookahead prefetching with trace computation in a scheme they called dependence graph precomputation. In this method, dependency dependencies are kept for a fixed number of most recent instructions in a circular buffer. The decoding mechanism feeds this buffer. As loads are decoded and added to the buffer, a process in hardware initiates which adds the load, then the instruction on which the load is dependent, then the instruction on which the new instruction is dependent, at a rate of one per cycle. While there may be more than one dependency in an instruction, only the first is used. Once the dependency list is complete or reached its maximum size, it is entered into a buffer for the Precomputation Engine (PE). The PE continually services valid lists one at a time by executing them with a separate single-issue execution unit. The problem is that register
values used may not be correct, as the PE is not synchronized with the primary execution unit. Across nine benchmarks, they improved performance by 13% and came to within 15% of perfect data cache performance. It would seem, however, that the number of active dependency chains being serviced and the average execution time of an innermost loop would greatly affect the effectiveness of this method.

2.4.2.6 Hardware Instruction Prefetching

Lookahead prefetching for data typically use tables of previously-encoded instructions in order to avoid having to prefetch instructions to look ahead. The notable exception to this is the method proposed by Lee et al. [78]. As an architecture's execution rate is bound by the ability of the Fetch Unit to supply instructions, instruction prefetching is becoming more and more necessary. For this reason, several schemes have been proposed for instruction prefetching.

Smith and Hsu [124] proposed a scheme which combined OBL and control-flow target prediction. In this scheme, the next I-cache line is always prefetched, and the target line is also prefetched if prediction indicates that the target will be taken. Their results showed that the benefit from the hybrid scheme was roughly equal to the sum of the benefits from each approach separately. Pierce and Mudge [102] argued, however, that this was due to the very small cache sizes used in Smith and Hsu's experimentation. They proposed prefetching both the next line and the target always, which means they always predict both the correct path and the wrong path. This technique, called wrong path prefetching, reduced execution times in their experimental framework by as much as 20%. They argued that a great deal of benefit came from prefetching the target when the target was predicted as not taken because that target was eventually reached by the program counter in a great majority (75%) of cases. Lee et al. [76] also validated the use of next-line prefetching, when used with many different speculative fetch policies. In particular, they argued for a simple non-blocking cache which would implement a resume policy, one in which the fetch unit could continue to draw instructions from the correct path in the I-cache, despite a pending miss caused by a misprediction. For reasons of cost effectiveness and complexity, however, in cases where

37
the miss penalty was large (i.e. for modern processors, when instruction reuse in the L2 is low) and the branch prediction rate was bad (i.e., when executing non-scientific code), they argued instead for a pessimistic policy, in which no I-cache miss would be processed until the execution unit could determine that the path is correct.

Reinman et al. [109] proposed fetch directed prefetching, which is similar to the lookahead in Baer and Chen’s work [10, 24, 26], except that the lookahead is powered by instruction prefetching instead of previously decoded instructions. Consequently, the I-cache is filled but the D-cache is not. Their scheme utilized a decoupled branch predictor [107], which allowed the lookahead unit to proceed even during I-cache misses. Their results were good, yielding 25% to 40% speedup on their selected benchmarks, although they also caused a higher increase in bus utilization than the next-line prefetching and stream buffer prefetching to which they compared themselves. Additionally, like Baer and Chen’s method, when a branch prediction proves false, it invalidates the queued prefetch requests, and the predictor must start over.

2.4.2.7 Irregular Hardware Data Prefetching

Most hardware data prefetching schemes detect irregular accesses specifically so that prefetching for these loads can be avoided. Roth et al. [114] argued that many of these so-called irregular accesses are references to fields within linked data structures, and therefore are somewhat predictable. Unlike Luk and Mowry’s work [84], the authors advocated hardware-based prefetching for these data structures, which necessitates that the inter-instruction load dependencies be determined at runtime instead of compile-time. They build a table of potential address producers, the PPW, which lists recently loaded data and their associated instruction addresses. If and when a load commits which uses a datum generated by a previous load as a base address, the new load is entered into the CT table as a consumer. Then prefetches can be generated for consumers when a producer is encountered. Instructions which consume the data they produce (e.g., the traversing pointer variable in a linked list traversal) are given the highest priority, and stack-based and constant loads are ignored. Prefetches generated by an associative lookup in the CT table are queued to the Prefetch
Request Queue (PRQ) and wait for unused data ports. They achieved roughly 0% to 25% speedup for pointer-intensive benchmarks.

Collins et al. [31] used a pointer cache, which is a simple mapping of load target address to load target value, in which both the target address and target value match viable heap pointers. Matching committed stores feed the pointer cache, and loads query the pointer cache in parallel with the D-cache. With this structure, they were able to perform limited load value prediction, which yielded results 50% greater than simple stride prefetching for a set of irregular benchmarks. By also prefetching two blocks from the heap, starting with the load value taken as a heap pointer and triggered on a hit in the pointer cache, their results increased by another 5%. The size of two blocks was arbitrary but was picked as an estimate for the size of an entire data structure referenced by the heap pointer matching a load target value.

2.4.2.5 Prefetching for Trace Caches

Jacobsen and Smith [56] applied the idea of hardware prefetching to trace caches, in a method called trace preconstruction. The method relies on the fact that while the execution continues to hit in the trace cache, the trace construction hardware (which includes a traditional branch predictor and I-cache) goes unused. The hardware can be used speculatively to build traces which may be needed in the future. The speculation is relatively simple; when a backwards branch or a subroutine jump is followed, traces are built speculatively starting with the branch-off point. In the case of backwards branches, this should capture the end-of-loop case, and in the case of subroutine jumps, this should capture the return trace. Since during preconstruction execution may encounter another of these conditions, a stack is maintained to keep track of all possible trace starting points and to give them priority. Cache lines prefetched for trace preconstruction are kept in fully associative prefetch buffers, and multiple traces may be serviced simultaneously. They found that of the SPECint benchmarks with large working sets, trace preconstruction was able to reduce misses by 30% to 80%, and when combined with trace preprocessing, yielded an average speedup of 14%.
2.4.2.9 Prefetching for the TLB

Saulsbury et al. [117] proposed prefetching specifically for the TLB unit, targeting conflict misses as victim caches do. In their scheme, a complete LRU stack is kept, and the prediction is made that if level $n$ is referenced, then the next level referenced will also be $n$. Since the TLB uses a LRU replacement policy, the top levels of the stack reside in the TLB itself; the remaining levels are kept in main memory. When a miss in the TLB occurs, and the access can be found in a low level of the LRU stack, the LRU stack is rotated to keep up-to-date, and the new TLB entry corresponding to that level of the LRU stack is prefetched. This approach cut the TLB miss rate in half. The study is noteworthy here in that it is essentially an extension to the idea of the victim cache and an application of the idea outside the study of keeping the L1 cache populated with useful data.

2.4.2.10 Hardware Prefetching Summary

Przybylski [105] argued against hardware prefetching approaches in 1990, citing memory bandwidth limitations and temporally clustered cache misses as obstacles against the effectiveness of prefetching methods. Also, Burger et al. [17] predicted in 1996 that memory bandwidth limitations (as opposed to latency) would dominate future computer systems. The research performed above, however, suggests that there are gains which prefetching can exploit if it is done intelligently enough. Indeed, Pulacharla and Kessler [88] argued that the right hardware prefetching approach could even be cost effective. And while memory bandwidth is increasingly a system bottleneck, this is all the more reason to make sure it is utilized to its fullest extent.

2.4.3 Combined Approaches

Although the hardware approaches discussed above do not exclude possible coexisting software approaches, Chen and Baser [26] followed up on their previous work [10, 24], studying both hardware and software prefetching in a shared memory multiprocessor architecture using an interconnection network. They found that software schemes were more conservative about increasing memory traffic, while hardware schemes were more efficient in instruction
execution. For their own hardware prefetching scheme, they found that a big part of the remaining memory latency is hit-time for prefetches which were not initiated early enough. This was caused by the limitation of tying the data prefetching to branch prediction; the lookahead unit can never break out of a loop without being reset, and in the benchmarks Matmat and Water there are many nested loops with only a few inner iterations. This prevents the LA-PC from getting adequately ahead of the PC in those loops.

They outlined a scheme incorporating hardware and software approaches. Their idea was to allow a hardware stride prefetcher to act freely on regular array accesses, and to support prefetching instructions to handle accesses to various-sized data objects. This would allow the hardware to operate in the regions in which it performs well without the extra instruction overhead of software prefetching, and this would be complemented with prefetch instructions only for the references that the hardware would miss. For the combined approach, they did not unroll and split, nor insert prefetch instructions for, innermost loops. These steps were taken, however, in testing software prefetching alone. The combined approach proved superior to either approach alone, for each of the four benchmarks studied. Between 78% and 90% of cycles lost to memory latency were recovered.

2.5 Proposal

Software prefetching is cheap to implement in hardware, but it introduces overhead in the program and requires that the compiler technology be present to exploit the hardware. Hardware prefetching, by contrast, allows execution time speedup without external assistance. Software prefetching is best relegated to an assistant of a hardware prefetching scheme, as Chen and Baer proposed [25]. The types of prefetching instructions to include in a processor design is a small question, but the form of the hardware prefetching to include is not. The best way to perform hardware prefetching remains an open question, especially in light of scalability problems with the most successful approach to date, that of Baer and Chen's lookahead-based data prefetching [10, 24].
Current approaches either ignore instruction prefetching, link it implicitly to data prefetching, or offer schemes which do not stay far enough ahead of the current PC to substantially decrease the average fetch latency. We propose that instruction prefetching can significantly improve program performance, particularly on large programs without a regular structure, and therefore it is worth performing. We also propose that the nature of instruction requests is sufficiently different from data requests to warrant a different prefetching approach. Finally, we propose that an instruction prefetching mechanism needs to remain far enough ahead of the current PC to allow adequate time for instructions to arrive in the I-cache. Therefore, in this dissertation, a prefetching approach is proposed which separates data prefetching from instruction prefetching and introduces an original method of instruction prefetching. Like many methods proposed, this method will place demands on the bandwidth requirements of the system; however, it will better utilize this bandwidth, and it will not greatly affect the critical path of the processor.
CHAPTER III

HARDWARE SUPPORT

This chapter presents the design details for the two hardware units, the Instruction Prefetch Unit and the Load Target Buffer, which will provide hardware support for instruction and data prefetching, respectively. For each unit, the hardware structure and its operation will be described, and its design parameters will be discussed. Those parameters are tested in Chapter 4.

3.1 IPU

The Instruction Prefetch Unit prefetches instructions by traversing the calling tree of the executing program. It must be able to process more instructions per cycle than the machine’s Fetch Unit in order to get ahead, and it has its own bus, the IPUBUS, to the I-cache. See Figure 3 for a diagram representing the IPU’s place in the processor, and Appendix C for a schematic diagram representing its interior design.

The IPU relies on the fact that modern superscalar processors typically can fetch and decode instructions faster than they can be executed. Unlike the IPU, the processor is limited by data dependencies and branch instructions. The IPU ignores branch instructions, except for procedure calls and returns. This means that the IPU can occasionally make a mistake in predicting procedural control flow. However, the IPU has a mechanism to detect and recover from such mistakes.

The I-cache is non-blocking and can handle a request by the Fetch Unit and one by the IPU in a single cycle, although it gives miss priority to the Fetch request, pulling needed words from memory (or an L2 cache) to satisfy that request before it does so for the IPU’s request.

In order to accomplish this, the IPU keeps internally:

- *state* - a 2-bit state (IDLE, RECEIVING, WAIT)
• ipPC - the current prefetch address.

• stack - a 256-entry stack of 3-word entries: src, the source of a subroutine call, dest, the destination of a subroutine call, and diff, the cumulative number of words from subroutine starts until calls to the next level. The Fetch Unit is assumed to be equipped with a return address stack (RAS) in the same format.

• Instagreement - the highest stack level where the IPU stack agrees with the processor’s call stack.

Under normal circumstances, the IPU will request a cache line from the I-cache and then scan the instructions in parallel for a subroutine call or subroutine return. While a request is outstanding, state is set to RECEIVING. If a request misses in the I-cache, the IPU uses the next cycle to request the next cache line. It does so assuming it will need the line; if not, the line is discarded upon arrival. The IPU requests at most one additional line; it does not repeat this process if the second line also misses.

When a line arrives and is scanned, if a subroutine call is seen, an entry is pushed onto the IPU’s stack. The immediate diff value is computed as the difference between the current ipPC and the start of the current function, which may be found at the top of the dest stack. This immediate diff value is added to the cumulative value at the top of the diff stack to yield the new cumulative value for diff. This value is then pushed onto the diff stack. At
the same time, \texttt{ipPC} is pushed onto the src stack, and the target of the subroutine call is pushed onto the dest stack. Finally, \texttt{state} is set to IDLE, which will trigger a new request at the beginning of the next cycle. The IPU ignores any unused words from its current request. If more calls are seen in the words received that cycle, only the first is taken at this time.

If a subroutine return is seen, an entry is popped off of the stack, and \texttt{ipPC} becomes src+1. \texttt{state} becomes IDLE, and the IPU will resume prefetching at the beginning of the next cycle.

This is the general approach, although there are a few details which still need to be addressed:

1. Even with this scheme it is possible for the Fetch Unit to get ahead of the IPU; it may branch over a call to a subroutine which the IPU is exploring. Rather than have the IPU stubbornly continue this way, at the first miss, the I-cache will signal the IPU, which will then reset to the current state of the Fetch Unit by copying the Fetch Unit’s RAS, setting \texttt{ipPC} to the current 	exttt{PC}, setting \texttt{lastagreement} to point to the highest stack level, and setting \texttt{state} to IDLE.

2. Since the IPU does not execute instructions, if it were to encounter a recursive function, it would continue to recurse without a terminating condition. Therefore, when the IPU encounters a subroutine call, it checks the destination address against its top 8 destinations in stack. Only if there is not a match does it descend into the subroutine.

3. At present the IPU does not attempt to follow indirect function calls, since the target of these calls would not be available. Instead, it ignores such calls. If the Fetch Unit follows an indirect call and subsequently triggers an I-cache miss, the resetting of the IPU will put it back on track again.

4. When, on a function return, popping the return address causes a stack underflow, \texttt{state} becomes WAIT. The IPU remains in this state until an I-cache miss causes a reset and sets \texttt{state} to IDLE.
5. By the same token, on a stack overflow the stack pointer increases but the contents of the stack itself remain unchanged. When a pop occurs on an overflowing stack, the stack pointer is decreased but a NU value is returned. While the IPU’s stack is overflowing, the IPU behaves as it does in a WAIT state. While the Fetch Unit’s stack is overflowing, it waits on the execution of the subroutine return instruction to obtain the new PC.

6. Another potential problem is that of the ipPC getting too far ahead of the PC and prefetching cache lines which displace those which the Fetch Unit will need. In order to avoid this problem, a value called ahead is calculated which represents the distance ahead of the ipPC. When the IPU would normally issue a prefetch request, it first compares ahead against an internal threshold. Only if ahead is less than threshold may the prefetch proceed in that cycle.

3.1.1 Calculating ahead

There are many possible ways of calculating ahead, but this dissertation will limit its investigation to three different methods, called uninformed, informed, and incremental ahead calculation. All of these methods yield differences which only reflect procedural control flow, as will be explained below.

3.1.1.1 Uninformed ahead calculation

In this method, ahead is calculated with knowledge obtained exclusively at runtime. The number calculated is not an accurate measure of how far ahead the ipPC is with respect to the PC, but it has the advantage of simpler implementation than informed calculation, and still provides a general approximation of the distance, as is shown below.

Although ahead is calculated incrementally (by notification of procedure calls, returns, and jumps), the number is a reflection of the current call stack of the Fetch Unit and IPU, as well as the ipPC and PC. For this reason, it is helpful to look at an example

46
of the program execution state. Figures 4-6 show the Fetch Unit's and IPU's stacks at one point in a program's execution. In Figure 5, the gray-shaded regions indicate the instructions that contribute to \textit{abend}. Each region represents a range of instructions. For example, in \textit{f1}, there exists a range from \textit{j1} to \textit{j2}. The contribution of this range is \textit{j2} - \textit{j1} = 3. In this figure, the end blocks are half-shaded to indicate that the range is three instructions, shaded as two whole blocks and two half blocks on the ends. In Figure 6, only the additions to \textit{diff} are shown at each level; recall that the numbers stored are actually cumulative (for simplified calculation). The top \textit{diff} on the IPU stack would therefore be (ipPC - \textit{f2}) + (j2 - f2) + (j2 - f1).

The total difference between \textit{PC} and \textit{ipPC} is (j2 - j1) + (PC - f4) + (j2 - f2) + (ipPC - f3). This number can be obtained from the IPU stack, the Fetch stack, and \textit{lastagreement}. The reason for keeping the \textit{diff} stack cumulative is so that a range of diffs can be added together by subtracting a lower stack entry from a higher one. In
Figure 5: Example execution state for uninformed instruction prefetching

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>dest</td>
<td>diff</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ipPC</td>
<td>f_1</td>
<td>(ipPC - f_1)</td>
<td></td>
</tr>
<tr>
<td>j_1</td>
<td>f_2</td>
<td>(j_1 - f_2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Example stack representations for uninformed instruction prefetching

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>dest</td>
<td>diff</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>f_1</td>
<td>(PC - f_1)</td>
<td></td>
</tr>
<tr>
<td>j_1</td>
<td>f_2</td>
<td>(j_1 - f_2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
this case, lastagreement points to the bottom entry, so the divergent diff is on the IPU stack are ipPC \(-f_3\), \(j_3 - f_3\), and \(j_2 - f_3\), which can be quickly added together by subtracting IPU_{diff[lastagreement]} (in the example, 0) from IPU_{diff[top]}. The same is done for the Fetch stack, and the total is \((ipPC - f_3) + (j_3 - f_3) + (j_2 - f_3) + (PC - f_3) + (j_1 - f_1)\). This does not match the shaded regions, because this number includes the distances from the start of function \(f_1\) to the jumping point \(j_1\) and \(j_2\), rather than the difference from \(j_1\) to \(j_2\). This problem can be fixed by subtracting the distance from \(f_1\) to \(j_1\) for each of the two times it was counted by the cumulative diff stack. In general this entire desired distance can be represented as follows:

\[
\text{ahead} = \text{IPU}_{\text{diff[top]}} + \text{FU}_{\text{diff[top]}} - 2\text{IPU}_{\text{diff[lastagreement]}} - 2(\text{FU}_{\text{src[lastagreement]}} + 1) - \text{FU}_{\text{dest[lastagreement]}},
\]

(7)

Increasing possession of either the PC or the ipPC inside a function will cause \text{ahead} to increase, and so will following procedure calls into other functions. Returning from procedure calls (by either the Fetch Unit or the IPU) will decrease \text{ahead}.

3.1.1.2 Informed ahead calculation

The only serious problem with uninform calculation is that in the Fetch Unit’s call stack, the progress of PC increases the value of \text{ahead} instead of decreasing it. Instead of adding the distances from the beginning of the functions to the jump points, the real numbers to add are the distances from those jump points to the end of the functions.

Figure 7 shows the same example as Figure 5, although this time the difference on the Fetch side is calculated from the PC to the end of the function. These values on the Fetch diff stack are still cumulative (see Figure 8), but if \text{ahead} is calculated this way, an increase in PC actually reduces the value of \text{ahead}, since the PC is gaining on the ipPC.

Deriving a formula for \text{ahead} using this method is done in the same manner as for Equation 7, although in this case instead of subtracting the difference \(j_1 - f_1\) twice, \(j_1 - f_1\) is subtracted once for the IPU side, and \(j_1 - j_1\) is subtracted once for the Fetch side. The latter can be remedied by subtracting the lastagreement + 1 diff entry on the Fetch stack instead of the lastagreement entry. The formula, then, is as follows:
Figure 7: Example execution state for informed instruction prefetching

Figure 8: Example stack representations for informed instruction prefetching

\[
\text{ahead} = \text{IPU}_{\text{diff}(\text{top})} + \text{FU}_{\text{diff}(\text{top})} - \text{IPU}_{\text{diff}(\text{lastagreement})} - \\
(\text{FU}_{\text{diff}(\text{lastagreement} + 1)} - \text{FU}_{\text{dest}(\text{lastagreement})}).
\]  

Informed ahead calculation is also done at runtime in hardware, but it needs the compiler to inform the mechanism of the endpoints of the functions on the call stack. This can be done by encoding each function length in the word immediately preceding that function’s first instruction. In this way, when a function is called, its length can be retrieved from memory at the same time as its first instructions.
Both uninformed and informed instruction prefetching use calculation which is based solely on the values of PC, ipPC, and the call stacks of the IPU and the Fetch Unit. Neither captures incremental history, or how those values change over time; each calculation method instead captures the static distance from one state (PC, IPU stack) to another (ipPC, Fetch Unit stack). In the example above, neither calculation includes the code in \( f_3() \), since executing that code is not, strictly speaking, required to transition from one state to the other. In practice, however, the IPU will have followed \( f_3() \) to reach its present state, and subsequently requested the necessary cache lines. The Fetch Unit may or may not follow \( f_3() \) as well.

An incremental approach notes only changes to intra-procedural control flow, and updates ahead incrementally every clock cycle, as shown in Figure 9. Intra-procedural control flow means that changes due to function calls and returns are completely discounted. For example, if the PC increments by 1, follows a function return which decrements it by 512, and continues another 2 instructions, then \( \Delta PC \) for that cycle is 3. Similarly, branches are counted; if the PC increments by 1, jumps back to the beginning of a loop 30 instructions back, and then continues into the new loop iteration another 3 instructions, then \( \Delta PC \) for that cycle is -26.

This means that as the IPU follows \( f_3() \), ahead gets incremented with the function’s size, and when the IPU returns from \( f_3() \), the increments made to ahead remain intact. Figure 10 demonstrates this configuration. It looks like informed instruction prefetching, except that all of \( f_3() \) is also counted. The problem with incremental prefetching is when the PC skips over a function which the IPU has counted towards ahead. Since the stack state is not used to calculate ahead, the additional increments of \( f_3() \) remain in ahead. This error is only cleared when the Fetch Unit encounters an I-cache miss and triggers a reset in the IPU.
3.1.2 Alternate Configurations

The problem of L1 I-cache contention between the IPU and the Fetch Unit can potentially be solved without resorting to a threshold *ahead* value, by adopting a different cache configuration. Two such configurations will be discussed here and tested in Chapter 4. Both of those configurations strive to prevent IPU requests from altering the contents of the Fetch Unit’s cache.

3.1.2.1 Separate Prefetch Cache

In this configuration, a relatively small prefetch I-cache sits between the L1 I-cache and the L2 cache (see Figure 11). A miss in the L1 I-cache queries the prefetch I-cache in the same cycle. IPU requests query the prefetch I-cache directly, so prefetch misses do not affect the contents of the L1 I-cache. On the other hand, Fetch Unit misses will cause the queried line to be placed in the prefetch I-cache as well as the L1 I-cache. Because a Fetch Unit miss resets the ipPC to the PC, this behavior is desirable.

3.1.2.2 Protected Mode I-cache

In this configuration, every line in the L1 I-cache contains an ownership bit. Lines requested by the Fetch Unit are owned by the Fetch Unit and have this bit set. Unused lines and lines requested only by the IPU have this bit cleared. Fetching behavior is the same, except
Figure 11: Separate Prefetch Cache (PFC) configuration

that the Fetch Unit relinquishes a function's lines when the return instruction retires. This happens by sending a special command to the I-cache; the function beginning address and function ending address are converted to line indexes, and the I-cache clears the ownership bit on this range of lines simultaneously.¹

Prefetch requests are modified so that a request can now be denied if satisfying the request would mean displacing a line owned by the Fetch Unit. On the other hand, if the Fetch Unit issues a request to the I-cache which conflicts with an IPU-owned line, the Fetch Unit request proceeds and the IPU-owned line is displaced. If the Fetch Unit requests a line previously requested by the IPU, so that no replacement is necessary, then the only action taken is to set the ownership bit so that the Fetch Unit now owns the line. This transference of ownership only occurs in this way; the only time ownership bits are cleared is by special request of the Fetch Unit.

3.1.2.3 Prefetch Cache in Protected Mode

Note that these configurations are not mutually exclusive. The prefetch cache could operate in protected mode. This means that there would be an L1 I-cache which only the Fetch Unit could access, and misses in this cache would check a prefetch cache. Fetch Unit requests

¹Note that the ending line index may be less than the starting line index; in this case, the lines higher than or equal to the starting line index as well as the lines less than or equal to the ending line index are cleared.
to the prefetch cache would transfer ownership of those lines so that prefetching could not then displace them. Because the Fetch Unit would have an L1 I-cache of its own as well as protected lines in the prefetch cache, it is likely that the instruction prefetching operations would be starved from lines owned by the Fetch Unit.

3.1.3 Implementation Issues

In order to achieve a reasonable benefit from the IPU, the IPU and the Fetch Unit need to operate largely independent of one another. If the IPU does its job properly and the machine’s internal instruction buffers do not fill up, the Fetch Unit will need to access the I-cache every cycle, and the IPU will for moderate stretches of time, access the I-cache every cycle as well. For these reasons, the I-cache cannot be a normal single-ported cache. As the cost per transistor is decreasing in processor production, a fully dual-ported cache is a reasonable demand for the system in future microprocessors. However, Wilson et al. [137] have achieved nearly equivalent performance by boosting the effective bandwidth of a single cache port, and the techniques which they studied would also be applicable for cache-sharing between the Fetch Unit and the IPU.

3.2 Load Target Buffer (LTB)

The Load Target Buffer is similar in principle to a Branch Target Buffer (BTB), tracking a set of recent loads and making predictions about future loads. As addresses are calculated in the address calculation units they are given to the LTB, which creates entries for them if necessary. The LTB may then issue prefetch requests to the D-cache (see Figure 12).
Like branches in the BTB, loads in the LTB are tagged by instruction address. For each load instruction in the LTB, the most recent load address, a count, and up to three strides are kept. If the sequence of load addresses received for a given instruction are $A_0, A_1, A_2, \ldots, A_n$, then $\text{stride}_1$ represents $A_n - A_{n-1}$, $\text{stride}_2$ represents $A_{n-1} - A_{n-2}$, and $\text{stride}_3$ represents $A_{n-2} - A_{n-3}$.

While count is less than a constant $K$, no prefetching is done. This is to allow the first few references to pass by so that the prefetching mechanism can be more confident about the regularity of the access pattern before it predicts future addresses. When count reaches or exceeds $K$ and if the access pattern is deemed regular, then the $N$th address ahead is predicted and prefetched ($N$ is another constant like $K$ whose specific value is discussed in Section 3.2.3).

The organization and stride prediction of the LTB are functionally equivalent to those of Baer and Chen's RPT [10]. The constants $K$ and $N$ are generalizations of the RPT's operation, but the primary difference between the two is the trigger by which predictions are made. In Baer and Chen's method, predictions are made from encountering load instructions with a lookahead unit. The LTB initiates prefetches at the same time its state is updated: when a load target address is calculated by the address calculation unit. It therefore can make predictions earlier than the RPT; when the load address in a loop's second iteration is calculated, the LTB will calculate the stride between the two addresses and prefetch $N$ iterations ahead, while the RPT must wait until the load instruction is next encountered by the lookahead unit. Even more importantly, the LTB does not have as severe a scaling problem as the RPT; there can only be as many accesses in one cycle as there are address calculation units. And even if a machine is fully superscalar, having as many address calculation units as decoding units, the LTB will be accessed only once out of every six instructions decoded.

3.2.1 Predicting the Nth address

Regular array access patterns will have stride patterns of the form $\ldots, a, a, b, a, a, \ldots$, where $b$ may change value from one appearance to the next. The $N$th address is predicted.
the current, then, is \( A_{\text{current}} + a \times N \). This prediction will be wrong when the next stride is actually a \( b \) value instead of \( a \), but this scheme will stick to predicting \( a \) and not be sidetracked by the occasional exception. To make sure that it sees a regular access pattern, it stores the last three strides (in \( \text{stride}_1 \), \( \text{stride}_2 \), and \( \text{stride}_3 \)), and requires that two of them consecutively match. The matching value is taken to be \( a \). Both Baer and Chen’s [10] and Eckemeyer and Vassiladis’s stride prediction work this way as well.

3.2.2 Implementation Issues

The operation of the stride prediction mechanism was discussed using the three most recent strides for clarity. Baer and Chen’s mechanism [10] illustrates how to accomplish the same predictions using only the most recent matching stride and two bits of state. The use of three full strides is discussed and simulated, but an implementation would be able to reduce amount of state required per load instruction.

3.2.3 Choices for \( K \) and \( N \)

The best selections for \( K \) and \( N \) will depend on the nature of the executing code. Selecting too large a value of \( K \) will inhibit prefetching for recently-established patterns, and selecting too small a value may allow erroneous prefetching for irregular patterns. As the number of total accesses increases much beyond \( K \), however, the specific value chosen for \( K \) becomes less important, while the value chosen for \( N \) becomes much more important.

The best value of \( N \) for any given load instruction takes into account the dynamic frequency of the instruction (i.e., how often it occurs in cycles) and the average memory latency for the load. Too small a value of \( N \), and data will not be prefetched early enough; too large a value of \( N \), and the prefetching system will spawn conflict cache misses.

The values of \( K \) and \( N \) can be static, programmable, or encoded into load instruction bits by the compiler. Since the the dynamic frequency and average memory latency are different for each load instruction, this last option has the greatest potential. However, there is a tradeoff in sacrificing instruction bits in order to encode extra information. In the case of MIPS indirect load instructions, the address is computed by adding the value of a specified integer register to an encoded offset. In the instruction encoding, 6 bits are
used as an opcode, 5 bits are used for the source register, 5 bits are used for the destination register, and 16 bits are used for the offset, as shown in Figure 13.

The size of the offset encoding allows numbers in the range -32768 to 32767. In practice, however, this range is quite generous; offsets are often zero [9], and tend to remain small. Figure 14 shows a histogram of bit requirements of load offsets, as taken from the compiled benchmarks. All 16 bits are rarely needed; in fact, 99.5% of the time, 13 bits or less are needed.

A set of indirect loads with prefetching hints can therefore be introduced with a minimum of intrusion. The same opcodes can be used as for regular loads; the only difference is that the three bits used to encode the prefetching hints will always be 000 for regular loads, whereas they will indicate one of 7 different prefetching configurations for the indirect prefetching loads. Since the value of N should matter much more than that of K, the best use of the bits should be to indicate one of 7 values of N, and to let K remain fixed.
Before conversion:

\[ \text{id } r1, 20000(r2) \]

After conversion:

\[ \text{addi } r3, r2, \#20000 \]
\[ \text{id } r1, (r3) \]

**Figure 15:** Conversions for loads with large offsets

In the cases where the load offset will not fit in a 13 bit encoding, a conversion is necessary to calculate the load address separately, to store the load address in another register, and to use the calculated address, as shown in Figure 15. For a reduction in offset encoding from 16 bits to 10 bits, this occurrence is rare enough to justify the added instruction with its potential increase in execution time.

3.2.4 Calculating N

If \( N \) is to be encoded into instruction bits for each load, it must be calculated by the compiler. Assuming a constant dynamic frequency \( F \) and a constant memory latency \( L \), this distance should be expressed as \( N = FL \). Since the benefit of data prefetching will largely be restricted to loads performed in loops, then \( F = \frac{1}{E_{\text{loop}}} \), where \( E_{\text{loop}} \) is the execution time of the instruction’s innermost loop. The algorithm to calculate \( N \) will make two simplifying decisions:

1. Each load instruction not contained in a loop (within its function) will not be prefetched.
2. If a load instruction’s loop itself contains another loop, \( N \) will be estimated as 1.

This leaves load instructions found in innermost loops. Rather than predict the cache hit rate of each load instruction, the algorithm will take a pessimistic view and assume that \( L \) is the maximum memory latency.

Calculating \( E_{\text{loop}} \) poses a problem, as an innermost loop may have any number of distinct traces, or instruction paths. For example, the innermost loop shown in Figure 16 has three
distinct traces: [A-B-G], [A-C-D-F-G], and [A-C-E-F-G]. The algorithm will therefore make two simplifying assumptions for the sake of tractability:

1. During program execution, only one trace per loop is used.

2. Branch directions have equally-weighted probability.

In the example, this means that the execution will either be [A-B-G] looped, [A-C-D-F-G] looped, or [A-C-E-F-G] looped, with no interspersing of these traces. Each trace does not have equal probability, however. Instead, the decision at basic block A is 50% for B and 50% for C. Therefore, the probability for trace [A-B-G] is 50%, while [A-C-D-F-G] and [A-C-E-F-G] share the other 50%, equally weighted at the decision at basic block C to yield 25% each.

Figure 16: An example of an innermost loop with multiple traces
$E_{loop}$ is therefore expressed as an expected value:

$$E_{loop} = \sum_{i} E_{T_i} \cdot p_{T_i}$$

(9)

Here, $E_{T_i}$ represents the execution time of trace $T_i$, and $p_{T_i}$ represents the probability that trace $T_i$ will be the one executed. $N$, then, can be expressed by the following:

$$N = \frac{L_{max}}{\sum E_{T_i} \cdot p_{T_i}}$$

(10)

In order to calculate $E_{T_i}$ for trace $T_i$, an iterative algorithm is used. The decode buffer is assumed to be filled with the instructions from an infinite number of instances of the trace. A grid is kept for cycles versus functional units. The algorithm schedules instructions for the current iteration $n$ on available functional units, placing the beginning of execution as early as possible while obeying data dependence constraints. When all instructions for iteration $n$ have been scheduled, the total execution duration for all $n$ iterations, represented by $E_{T(x \times n)}$, is noted. Then the next iteration is scheduled; this will normally overlap execution of the previous iteration. As more iterations are scheduled, the average execution time, calculated as $\frac{E_{T(x \times n)}}{n}$, converges towards a single number, $E_{T_i}$. The proof of this hinges on the fact that there exist upper and lower bounds on $E_{T(x \times (n+1))} - E_{T(x \times n)}$, the increase in execution time for iteration $n + 1$, as shown:

$$\Delta_{max}(E_{T_i}) \leq E_{T(x \times (n+1))} - E_{T(x \times n)} \leq \Delta_{max}(E_{T_i})$$

(11)

The largest execution time increase between iterations occurs during the scheduling of the first iteration, when there is no previous iteration with which to overlap execution:

$$\Delta_{max}(E_{T_i}) = E_{T(x \times 1)}$$

(12)

Likewise, the smallest increase possible is during the second iteration, when overlap is possible but when there are the fewest iterations with which to contend:

$$\Delta_{min}(E_{T_i}) = E_{T(x \times 2)} - E_{T(x \times 1)}$$

(13)

From Equation 11 comes:

$$E_{T(x \times n)} + \Delta_{min}(E_{T_i}) \leq E_{T(x \times (n+1))} \leq E_{T(x \times n)} + \Delta_{max}(E_{T_i})$$

From Equation 11 comes:

$$E_{T(x \times n)} + \Delta_{min}(E_{T_i}) \leq E_{T(x \times (n+1))} \leq E_{T(x \times n)} + \Delta_{max}(E_{T_i})$$

(10)
\[ \lim_{n \to \infty} \left( \frac{E(T_{1}(x,n)) + \Delta_{\text{min}}(E_{T_{1}})}{n + 1} \right) \leq \lim_{n \to \infty} \left( \frac{E(T_{1}(x,n+1)) + \Delta_{\text{min}}(E_{T_{1}})}{n + 1} \right) \leq \lim_{n \to \infty} \left( \frac{E(T_{1}(x,n)) + \Delta_{\text{max}}(E_{T_{1}})}{n + 1} \right) \] (14)

Since \( \Delta_{\text{min}}(E_{T_{1}}) \) and \( \Delta_{\text{max}}(E_{T_{1}}) \) are finite and \( n \) converges to \( n + 1 \):

\[ \lim_{n \to \infty} \left( \frac{E(T_{1}(x,n))}{n} \right) \leq \lim_{n \to \infty} \left( \frac{E(T_{1}(x,n+1))}{n + 1} \right) \leq \lim_{n \to \infty} \left( \frac{E(T_{1}(x,n)) + \Delta_{\text{max}}(E_{T_{1}})}{n} \right) \]

The algorithm then, watches for the convergence of the upper and lower bounds:

\[
\begin{align*}
\text{lbou}\text{nd} &\leftarrow \frac{E(T_{1}(x,n)) + \Delta_{\text{min}}(E_{T_{1}})}{n + 1} \\
\text{ubou}\text{nd} &\leftarrow \frac{E(T_{1}(x,n)) + \Delta_{\text{max}}(E_{T_{1}})}{n + 1}
\end{align*}
\]

Since this may require an infinite number of iterations, integer division is employed, which changes the actual calculations to the following:

\[
\begin{align*}
\text{lbou}\text{nd}_{\text{int}} &\leftarrow \frac{E(T_{1}(x,n)) + \Delta_{\text{min}}(E_{T_{1}})}{n + 1} \\
\text{ubou}\text{nd}_{\text{int}} &\leftarrow \frac{E(T_{1}(x,n)) + \Delta_{\text{max}}(E_{T_{1}})}{n + 1}
\end{align*}
\]

Note that \( \text{lbou}\text{nd}_{\text{int}} \) and \( \text{ubou}\text{nd}_{\text{int}} \) will never converge if \( E_{T_{1}} \) is actually an integer.

For this reason, the algorithm allows for the discrepancy and terminates when \( \text{lbou}\text{nd}_{\text{int}} \geq \text{ubou}\text{nd}_{\text{int}} - 1 \). Then \( E_{T_{1}} \) is taken to be \( \text{ubou}\text{nd}_{\text{int}} \).

The value of \( N \) is rounded up from the calculation expressed in Equation 10, as shown:

\[ N = \left\lceil \frac{L_{\text{max}}}{\sum_{j \in I} E_{T_{1}^j \! T_{1}}^j} \right\rceil \] (15)

Since the values of \( E_{T_{1}} \) are used only to calculate \( N \), an optimization of the algorithm is possible. When calculating any given \( E_{T_{1}} \) with a probability \( p_{T_{1}} \), it may be possible to determine \( N \) via upper and lower bounds.

\[ E_{\text{loop}} = \sum_{j \in I} E_{T_{0}}^j p_{T_{0}} + E_{T_{1}}^j p_{T_{1}} + \sum_{j \in I} E_{T_{1}^j \! T_{1}}^j \] (16)

Just before computing \( E_{T_{1}} \), the first term above is known, and Equation 14 gives lower and upper bounds for \( E_{T_{1}} \) after scheduling iteration \( n \). If \( \sum p_{T_{1}} = 0 \forall j \neq i \), the last term...
will be zero, so:

\[ \text{lbounds}_n(E_{\text{loop}}) = \sum_{j<i} E_{T_j} p_{T_j} + \frac{E_{T_i \cdot \text{init}} + \Delta_{\text{min}}(E_{T_i})}{n+1} \]

\[ \text{ubounds}_n(E_{\text{loop}}) = \sum_{j<i} E_{T_j} p_{T_j} + \frac{E_{T_i \cdot \text{init}} + \Delta_{\text{max}}(E_{T_i})}{n+1} \]

And:

\[ \text{lbounds}_n(E_{\text{loop}}) \leq E_{\text{loop}} \leq \text{ubounds}_n(E_{\text{loop}}) \]

From this, the upper and lower bounds of \( N \) can be determined:

\[ \left[ \frac{L_{\text{max}}}{\text{ubounds}_n(E_{\text{loop}})} \right] \leq N \leq \left[ \frac{L_{\text{max}}}{\text{lbounds}_n(E_{\text{loop}})} \right] \]

If, however, \( \sum p_{T_j} > 0 \ \forall j > i \), then \( E_{\text{loop}} \) is unbounded:

\[ 1 \leq N \leq \left[ \frac{L_{\text{max}}}{\text{lbounds}_n(E_{\text{loop}})} \right] \]

In either case, when computation converges on a single value, that value is taken as \( N \).
EXPERIMENTAL RESULTS

This chapter discusses the results of many experiments designed to test this prefetching method and its various parameters. First, the experimental setup is described, and then results are presented. The experiments are divided into two sections: hardware experiments, which test various hardware configurations for fixed software configurations, and software experiments, which test various software configurations for fixed hardware configurations. In each case, the experiment is described, the results are presented in graphical form, and both quantitative and qualitative analyses are given.

4.1 Experimental Setup

Experiments were conducted through the use of a detailed superscalar machine simulator, simulating execution of the SPEC95 benchmark suite.

4.1.1 Machine Simulation

The simulator, SuperDLX, is a parameterized, machine-level, superscalar simulator originally developed at McGill University. SuperDLX simulates DLX object code with a variety of superscalar machine configurations. Simulation of the proposed hardware in this document was accomplished by extending SuperDLX, and cache behavior was accomplished by extending Cachesim and linking it to SuperDLX. SuperDLX collects cycle counts along with cache and prefetch statistics and is able to determine the total latency for instruction fetching and reading and writing data and the total execution time. Machine-level simulation should be more accurate than trace-based simulation for testing memory hierarchy configurations, as trace-based simulations do not simulate the result on the memory hierarchy of fetching instructions and even data down mispredicted paths.

\footnote{The DLX instruction set is a variation of the MIPS instruction set.}
\footnote{Cachesim was originally written by Scott Wills.}
Table 1: Simulation parameters which differ from the R10000

<table>
<thead>
<tr>
<th>Parameter Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate load and store queues</td>
<td></td>
</tr>
<tr>
<td>A RAS of 256 entries, allowing the processor to return from functions even before the return address is computed</td>
<td></td>
</tr>
<tr>
<td>An LTB of 266 entries, organized as 64 4-way associative sets</td>
<td></td>
</tr>
<tr>
<td>A non-blocking 1 cycle L1 cache (split into a 32 Kb dual-port I-cache and a 32 Kb dual-port D-cache)</td>
<td></td>
</tr>
<tr>
<td>A 1Mb 6 cycle on-chip L2 cache with a 56 cycle, 128 bit access to memory</td>
<td></td>
</tr>
<tr>
<td>A 128 bit IPUBUS, a 128 bit L1 D-bus, and a 128 bit L1 I-bus</td>
<td></td>
</tr>
</tbody>
</table>

The basic machine configuration used is chosen to mimic the MIPS R10000 processor (refer to [88] for details) with the additions and/or changes listed in Table 1.

4.1.2 Cache Simulation

Cache simulation in SuperDLX was built by extending the stand-alone cache simulation program cachesim by Scott Wills. In order to more easily fit in with the other SuperDLX modules, the interface to the cache simulation module was constructed to behave in the same manner as the other functional units, such as floating-point multiplication units. Each instruction in the simulated machine is represented by a structure which includes the instruction address, the type of functional unit it requires to execute, and statistical information such as the cycles in which the instruction was fetched, decoded, and executed. When an instruction has been decoded and one of its required functional units is available, the functional unit latency is loosed up, and the instruction is marked to complete this many cycles into the future.

In the same manner, when a memory access is requested, the latency for the operation is returned, based on the activity of the memory system. To achieve the same effect in cache simulation, it was necessary to simulate both caches and buses and to keep track of when any given cache or bus will be available for a new request. For example, in a request to cache C for data to be passed along on bus B, first it is necessary to determine when cache C will be available to take the request, which may be immediately. Then C's access time

64
is incurred. If the request requires further memory operations, these are then performed, and the latency from these are also incurred. In fact, even on a hit, the requested line may not yet be available, because the line may arrive on a future cycle; in this case, the cache must wait for the line to arrive from the next level of the memory subsystem. The latency returned is the total elapsed time from the request cycle until the cycle in which the first transfer of the request is satisfied. A request of a size larger than $B$'s width will be completely satisfied in later cycles, but the first words returned will be immediately available. The total transfer time is determined by the request size $S$, $B$'s width $W$, and $B$'s transfer time $T$:

$$\text{total transfer time} = \left\lceil \frac{S}{W} \times T \right\rceil$$  \hspace{1cm} (20)

A value of $T$ equal to 1 simulates a pipelined or burst-mode cache, while a value of $T$ equal to the cache access time simulates a non-pipelined cache. Also, the value specified for the cache access time is meant to overlap the transfer time; i.e., if both are set to 1 for an L1 cache, then a hit in the cache requires only 1 cycle, not 2. Finally, the requested word is always in the first transfer, and then transfers continue in round-robin style until all words in the line are transferred. For example, if the line size is 8 words, the bus width is 2 words, and word 3 is requested, then the 4 transfers are, in order: words 2 and 3, words 4 and 5, words 6 and 7, and words 0 and 1.

The cache simulation algorithm does not keep track of all the request(s) it is currently satisfying, only the most recent request made and the next cycle for every cache and bus. Likewise, the state of each cache reflects what the state will be at the beginning of that cache's next free cycle. For example, on a read miss, the cache writes back the dirty lines it will replace, if any requests the new data, and calculates when this will be done. The cache only knows what the most recent request was and when the first transfer of that request will be complete. It is of no concern to the cache what requests for which instructions it will be satisfying from the present cycle until the next free cycle; the instructions themselves

---

Such as a miss during a read or a miss during a write in writeback mode.
know when their own memory requests will be completed, because this was determined at
the time they each made their requests.

Finally, since requests can be fulfilled between the current cycle and the next free cycle,
at which the cache contents will be current, it is necessary also to keep track of when
requested cache lines arrive. This enables the cache simulation algorithm to include the
proper waiting time in calculating latency in the case where a line is present at the start of
the cache's next free cycle but not yet available during the cycle the request is made. This
is implemented by adding two fields to the cache line structure: one reporting the cycle
during which the first bus width's worth of words will be available, and another indicating
which words will be first.

4.1.3 Benchmarks

The SPEC95 benchmarks were chosen as representative applications for their variety and
availability. The benchmarks written in C were compiled with gcc version 2.7.2, and the
benchmarks written in Fortran were converted to C using f2c, and then compiled with
gcc version 2.7.2. All the benchmarks were compiled at optimization level -O2 with no
other optimizations specified. Since the version of the DLX machine description obtained
included branch delay slots, gcc was further instructed to place nop instructions in the
branch delay slots rather than to reorder the instructions. Then a filtering program removed
these inserted nop instructions.

Due to the time required to simulate at a high level of detail, the benchmarks were
executed for a limited range of instructions. These limits were selected independently for
each benchmark by observing the point at which cache behavior progressed past the warmup
phase; see Section 4.1.4. Table 2 lists the benchmarks with their code size, the SPEC95
input configuration, and the simulation range used. Note that most of these benchmarks
do not fit into the 32 Kb I-cache. In fact, gcc's code segment does not even fit into the 1
Mb L2.

In addition, because a copy of gcc's like compiled for the DLX processor could not
be obtained before experimentation was well under way, the experiments do not include
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
<th>Configuration</th>
<th>Simulation Range</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sople</td>
<td>58 Kb</td>
<td>SPEC95 train input</td>
<td>149m - 249m inst.</td>
<td></td>
</tr>
<tr>
<td>sople</td>
<td>157 Kb</td>
<td>SPEC95 train input</td>
<td>259m - 485m inst.</td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>9 Kb</td>
<td>SPEC95 train input</td>
<td>0 - 514m inst.</td>
<td>whole</td>
</tr>
<tr>
<td>fpppp</td>
<td>67 Kb</td>
<td>SPEC95 train input</td>
<td>0 - 291m inst.</td>
<td>whole</td>
</tr>
<tr>
<td>gcc</td>
<td>1314 Kb</td>
<td>SPEC95 train input</td>
<td>0 - 1686m inst.</td>
<td>whole</td>
</tr>
<tr>
<td>gcc</td>
<td>295 Kb</td>
<td>SPEC95 train input</td>
<td>17m - 171m inst.</td>
<td></td>
</tr>
<tr>
<td>hydro2d</td>
<td>45 Kb</td>
<td>SPEC95 train input</td>
<td>750m - 1626m inst.</td>
<td>3 iterations</td>
</tr>
<tr>
<td>sipeg</td>
<td>127 Kb</td>
<td>SPEC95 train input</td>
<td>102m - 252m inst.</td>
<td></td>
</tr>
<tr>
<td>fl</td>
<td>68 Kb</td>
<td>SPEC95 train input</td>
<td>0 - 265m inst.</td>
<td>whole</td>
</tr>
<tr>
<td>m88k5im</td>
<td>138 Kb</td>
<td>SPEC95 train input</td>
<td>0 - 178m inst.</td>
<td>whole</td>
</tr>
<tr>
<td>mgrid</td>
<td>18 Kb</td>
<td>SPEC95 train input</td>
<td>944m - 1044m inst.</td>
<td></td>
</tr>
<tr>
<td>perl</td>
<td>264 Kb</td>
<td>SPEC95 train input</td>
<td>1119m - 1219m inst.</td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td>52 Kb</td>
<td>SPEC95 train input</td>
<td>283m - 383m inst.</td>
<td></td>
</tr>
<tr>
<td>sim</td>
<td>10 Kb</td>
<td>SPEC95 train input</td>
<td>325m - 534m inst.</td>
<td>3 iterations</td>
</tr>
<tr>
<td>tomone</td>
<td>5 Kb</td>
<td>SPEC95 train input</td>
<td>283m - 977m inst.</td>
<td>3 iterations</td>
</tr>
<tr>
<td>turk2d</td>
<td>54 Kb</td>
<td>SPEC95 train input</td>
<td>2123m - 4118m inst.</td>
<td>1 iteration</td>
</tr>
<tr>
<td>vortex</td>
<td>608 Kb</td>
<td>SPEC95 train input</td>
<td>621m - 721m inst.</td>
<td></td>
</tr>
<tr>
<td>wave5</td>
<td>150 Kb</td>
<td>SPEC95 train input</td>
<td>368m - 1273m inst.</td>
<td>1 iteration</td>
</tr>
</tbody>
</table>

† Due to the small size of the train buffers, 1000000-byte buffers were used instead.
‡ Only the input set `jumble` was used.

Simulation of libc functions. Instead, stub functions which execute traps were compiled in with the SPEC95 benchmarks. The trap handles the libc calls by having the simulator make the calls and translating the results back into the memory and register file of the simulated machine. Execution of a trap occurs during the commit stage, takes 1 cycle, and flushes the machine pipeline.

While it is unfortunate that the operating system could not also be modeled as part of the executing benchmarks, several studies [131, 136] have shown that the amount of time spent executing operating system code is small compared to the time spent executing application code. These results indicate that the results obtained by using traps for library procedures are qualitatively similar to those which would be obtained from more complete simulations.
Validation methods were used as spot checks against likely errors which would affect the experimental results. While not definitive, the methods do give reasonable assurance without requiring an disproportionate amount of work. The first method used verified simulation accuracy, that the simulated machine performed the same tasks as the real machine would, on a cycle-by-cycle basis. The second method used plots of the benchmarks' cache behavior and execution throughput in order to choose simulation durations, which were statistically significant representations of the benchmarks' overall behavior. Finally, cyclic redundancy checks (CRC checks) were performed to insure that for any given benchmark, simulations using different configurations always followed the same instruction path, i.e. that the same instructions were committed in the same order.

4.1.4.1 Simulation Accuracy

In order to validate simulation accuracy, the progress of every instruction was logged for several hundred cycles. This information was plotted by a program called analysis, which reads SuperDLX log files and produces a multi-page Postscript plot with program assembly instructions on the vertical axis, and cycle count on the horizontal axis. Figure 17 shows a sample of the output produced for a simulation run of the regular matrix multiplication benchmark daxpy. In each grid space, the status of the instruction for that cycle is expressed as a single character. For example, if the instruction is fetched on this cycle, the letter 'F' appears in this grid space. If the instruction is fetched on this cycle, and the instruction is being executed from a previous iteration during the same cycle, then both 'F' and 'E' appear in the grid space. In addition, true runtime data dependencies and runtime structural dependencies are plotted as arrows between 'E' spaces of relevant functions. In this way, the timing of instruction execution is given in a visual form, with an indication of why an instruction was delayed in executing.

In Figure 17 for example, the first iteration of the main loop is executing at cycle 2631, and the second iteration begins at 2642. The first iteration takes a relatively long time to execute, since the loop instructions were present in the L2 cache but not the I-cache.
This is the reason for the long duration between the fetch and decode stages (from cycle 2635 to cycle 2642 on instructions 1100-1103). At the beginning of the second iteration, however, activity increases greatly. The plot also shows that the execution of instruction 1098 is delayed by a cycle in the first and second iterations because the simulated machine contained only one floating point address calculation unit.

Plots were studied closely for *daxpy* and on a spot-check basis for a sample of other benchmarks, both with and without pre-fetching active. The sample plots indicated that the simulator was yielding correct timing information based on the machine configurations and benchmarks that it simulated.

### 4.1.4.2 Simulation Durations

Since many of the benchmarks took many hours to simulate for a single configuration, and since some experiments required hundreds of simulations per benchmark, benchmarks were only simulated for a fixed number of simulated committed instructions. Additionally, to avoid unfair results, statistics were taken only from a range beginning after any startup effects. The range of instructions was chosen for each benchmark by sampling average cache behavior at fixed intervals; see Appendix A. The benchmarks *hydro*, *swim*, *turbo*, and *wrec* exhibited periodic cache behavior, in which miss rates would change suddenly at the beginning of each outer loop iteration. These benchmarks were executed for a whole number of iterations after the first iteration. Other benchmarks exhibited a flattening of average read latency and average fetch latency curves. The range was chosen just past the elbow of these curves. At least 100 million instructions were simulated for each benchmark.

### 4.1.4.3 Instruction Path

The final method of validation was in ensuring that for each benchmark, the same instructions were committed in the same order regardless of the simulation parameters. This validation was necessary since the simulator used was a full machine simulator operating on static code rather than dynamic traces; a bug in the simulator code could cause the

---

*Turbo* required 10 days of processing time for a single complete simulation.

70
simulator to execute a different dynamic trace for the same input but different machine configurations, which would yield wildly inaccurate results. The dynamic trace was therefore verified by treating the sequence of committed instruction addresses as a binary data stream of 32-bit integers and performing a CRC check on the stream. A base case was established by executing the benchmark in non-pipelined mode (i.e., one instruction had to be committed before the next could be fetched) with a simulated perfect cache. Then all other simulations compared the computed CRC against that computed in the base case. This feature was helpful particularly in verifying simulator ports to different architectures with different endian encodings. In fact, the benchmark hydrohold yielded a different base case CRC on an Intel platform than on a Sun platform. The reason turned out to be a floating point computation that yielded 0.0 on one platform and a very small negative number on the other. A comparison to the result of this computation caused a branch to be taken in one case and not taken in the other case. As a result, all hydrohold simulations were performed on an Intel platform.

4.1.5 Metrics

Three metrics will be used to evaluate the methods explored in this document. The first of these is the IPC, the ratio of number of instructions committed to the number of cycles elapsed. This number is slightly lower than the IPC computed for the number of instructions executed, or for the number of instructions fetched. This figure is preferred, however, because it is not artificially raised by high branch misprediction rates. Also, the ratio of an experimental result’s IPC to a base result’s IPC for the same instruction count is equivalent to the speedup.

The second metric used is one proposed by Baer and Chen[10], called IPCM. IPCM is the IPC computed only for memory operations. Rather than using elapsed simulation cycles for overlapping memory operations, the IPCM reflects average memory latency. So, if a memory operation begins at cycle 10 and lasts 5 cycles, and another operation begins at cycle 12 and lasts 5 cycles, the IPCM would be 0.2, even though the span of operations only required 7 cycles instead of 10. Also, this figure reflects memory operations for instructions
regardless of whether the instruction is committed or not. For the example architecture used, all writes are issued during the commit stage, but loads are not committed until the memory result is available, so the load instruction could at times be canceled even after the memory operation had completed. Results will focus on the ratio of IPCM’s calculated only for load instructions.

The third metric is called recovery. Recovery looks at the memory problem in the following way: if prefetching works perfectly, then memory operations would always hit in the cache. If the hit time were 1 cycle, the average memory latency would be 1. A machine operating without prefetching may have an average memory latency of, for example, 2.7. This means that, on average, 1.7 cycles are lost per operation because of cache misses. If a prefetching method can recover all 1.7 “lost” cycles, then it is working at 100% efficiency. If, on the other hand, it can recover only 1.5 cycles, then it is working at 1.5/2.7 = 88.2% efficiency. If \( M_{base} \) represents the base average memory latency of 2.7, and if \( M_p \) represents the average memory latency with prefetching, 1.2, and if \( M_{hit} \) represents the hit latency of 1, then the efficiency is represented by:

\[
eff = \frac{M_{base} - M_{hit}}{M_{base} - M_{hit}}
\]  

(21)

When making these comparisons, it is unfair to include prefetch operations in the average \( M_p \), as the entire point of prefetching is to lower the average unmasked latency for the necessary operations so that instructions can be executed more quickly. A prefetching method which has a low average latency for the prefetching operations themselves is actually not performing well, as it is not often requesting new cache lines.

Since average latency is the average number of cycles required to complete a number of operations, \( M = \frac{1}{T_{PCM}} \). From this, recovery can also be expressed by:

\[
eff = \frac{\frac{1}{IPCM_{base}} - \frac{1}{IPCM_p}}{\frac{1}{IPCM_{base}} - \frac{1}{IPCM_{hit}}} = \frac{1 - M_{hit}IPCM_{base}}{1 - M_{hit}IPCM_{base}}
\]  

(22)

\(^b\)Meaning that the average latency is 2.7 - 1.5 = 1.2 cycles
Note that a recovery value of 0 means that the prefetching method has not changed the P'CM at all. A value of 1 means that the prefetching method is working optimally. And a negative value means that the prefetching method is degrading performance.

4.1.6 Comparison to Baer and Chen's Experimental Setup

Since the data prefetching approach described in this paper bears many features of Baer and Chen's approach [10, 24], the experimental setup used is summarized in the terms used by Baer and Chen in their papers: this experimental setup uses a hybrid prefetching model on an overlapped cache model with hardware register renaming.

4.2 Prefetching Potential

Each benchmark has a hard limit for memory-based speedup, as each requires a minimum number of cycles to execute in the CPU with the given machine configuration, even if every memory request hits in the cache. This limit was explored experimentally by comparing simulations using different combinations of perfect caches (i.e., caches which always contain the data requested).

In moving from a perfect to a realistic cache, the additional execution time is unmasked memory latency. First, the assumption was made that this unmasked memory latency had independent I-cache (I) and D-cache (D) components, as well as an L2 cache component divisible into instruction (L2i) and data (L2d) components.

When a benchmark is executed with a perfect I-cache, the reduction in execution time should reflect I as well as L2i, as shown:

$$E_{normal} - E_{perfect} = I + L2i$$  \hspace{1cm} (23)

Similarly:

$$E_{normal} - E_{perfect} = D + L2d$$  \hspace{1cm} (24)

If a benchmark is executed with a perfect I-cache and a perfect L2 cache, the reduction in execution time should reflect I and both L2i and L2d:

$$E_{normal} - E_{L2,perfect} = I + L2i + L2d$$  \hspace{1cm} (25)
Similarly:

\[ E_{normal} - E_{system} = D + L2_i + L2_d \]  

(20)

From Equation 23 and Equation 25, \( L2_d \) can be calculated:

\[ L2_d = (I + L2_i + L2_d) - (I + L2_i) \]
\[ = (E_{normal} - E_{system}) - (E_{normal} - E_{system}) \]
\[ = E_{system} - E_{system} \]  

(27)

Similarly from Equation 24 and Equation 26:

\[ L2_i = E_{system} - E_{system} \]  

(28)

Then it follows from Equation 23 and Equation 28:

\[ I = (I + L2_i) - (L2_i) \]
\[ = (E_{normal} - E_{system}) - (E_{system} - E_{system}) \]
\[ = E_{normal} + E_{system} - (E_{system} + E_{system}) \]  

(29)

Similarly from Equation 24 and Equation 27:

\[ D = E_{normal} + E_{system} - (E_{system} + E_{system}) \]  

(30)

Finally, CPU is merely the remaining execution time when all caches are perfect:

\[ CPU = E_{system} \]  

(31)

Note that the total execution time \( E_{normal} \) should be equivalent to the sum of its calculated components, as shown:

\[ E_{normal} = I + D + L2_i + L2_d + CPU \]
\[ = (E_{normal} + E_{system} - (E_{system} + E_{system})) \]
\[ + (E_{normal} + E_{system} - (E_{system} + E_{system})) \]
\[ + (E_{system} - E_{system}) \]
\[ + (E_{system}) \]
\[ = 2E_{normal} + E_{system} - (E_{system} + E_{system}) \]  

(32)
Figure 18: Composition of execution time for the int SPEC95 benchmarks

This assures that:

\[ 0 = E_{\text{normal}} + E_{\text{perf}} - (E_{\text{iperf}} + E_{\text{idperf}}) \]
\[ E_{\text{iperf}} + E_{\text{idperf}} = E_{\text{normal}} + E_{\text{perf}} \]  

(33)

Figure 18 shows the breakdown of execution time for the integer benchmarks, and Figure 19 shows the breakdown of execution time for the floating-point benchmarks. The components CPU, L2d, L2i, D, and I as defined and calculated above are represented, normalized by the experimental execution time \( E_{\text{normal}} \). When the assumption represented by Equation 33 is not true, the total normalized value will not be 100%.

Figure 18 shows that the only integer benchmark with a significant\(^6\) L2 component is perf, the integer benchmarks with significant instruction components are go, m88kplus, gcc, perf, and vortex, and the integer benchmarks with significant data components are go, gcc, compress, ls, and perl.

Figure 19 shows that the floating-point benchmarks with significant L2 components are tomcat, swim, sub2cor, and hydro2d, the only floating-point benchmark with a significant parameter.

---

\(^6\)defined as 5% or greater
Figure 19: Composition of execution time for the fp SPEC95 benchmarks

instruction component is \( fppp \), and all of the floating-point benchmarks have significant data components.

For the purposes of evaluating prefetching configurations, those benchmarks with significant memory components of the type tested are referred to as *prefetching candidates*.

### 4.3 Instruction Prefetching

In this section, the three flavors of threshold-based instruction prefetching are evaluated. Each of the given benchmarks was executed with the normal configuration above and with instruction prefetching using thresholds of powers of 2, until the threshold was high enough not to be reached. The experiments show that the simplest method (incremental instruction prefetching) works nearly as well as the other two methods, while having the advantage of being much easier to implement. Finally, incremental instruction prefetching is compared against protected-mode caches and separate prefetch caches, and proves superior to either on its own or both combined. Refer to the second claim in Chapter 1. Data prefetching for all of these experiments was disabled.
4.3.1 Uninformed Instruction Prefetching

Figure 20 shows the speedup of the candidate benchmarks' recovery in uninformed instruction prefetching versus threshold. The floating-point benchmark fppp delivers the strongest improvement of 30.3% at a threshold of 4096. This is due to regular procedural control flow, limited temporal locality, and a code size large enough to cause cache conflicts. By contrast, perl delivered a maximum speedup of only 2.8%, because of procedural unpredictability. The remaining instruction prefetching candidates yielded speedups between 5 and 10%.

Most of the benchmarks' performance begins to degrade around a threshold of 4096, although this sensitivity is strongest for fppp. It is at 4096 that the highest geometric mean speedup, 9.7%, is yielded. Note, however, that this value remains much the same for an order of magnitude lower and higher (i.e., 2048 and 8192).

Figure 21 shows the fetch cycle recovery rates for uninformed instruction prefetching and gives an indication of how well the instruction prefetching algorithm performs. The recovery rates are modest, peaking at 67.2% for fppp, which reduces the average fetch latency from 2.1 to 1.3. The most revealing aspect of this figure is that the recovery rate
curves look like the speedup curves, except that a different vertical scaling factor has been applied to each benchmark. For example, while the prefetching algorithm works nearly as efficiently for gcc as for fpkpp, fpkpp is much more sensitive to changes in average fetch latency than gcc. This is because gcc is able to overlap fetch latency with computation from past instructions more often than fpkpp, which can execute and commit instructions almost as fast as they can be decoded.

4.3.2 Informed Instruction Prefetching

Figure 22 shows the effect of threshold on the IPC of the candidate benchmarks when informed instruction prefetching is used. The most obvious and significant result here is that the speedups are very similar in amplitude and form to those when uninformed instruction prefetching is used, as shown in Figure 20. In fact, fpkpp peaks again at 4096 with a speedup of 29.6%, which is not statistically different than when uninformed instruction prefetching is used. The geometric mean peaks at 8192 instead of 4096, but the value, 9.6%, is nearly identical to that obtained by uninformed instruction prefetching.

Figure 23 shows the fetch cycle recovery rates for informed instruction prefetching.
Figure 22: IPC speedups for informed instruction prefetching

Note that these curves again are very similar in form to those for uninformed instruction prefetching, as shown in Figure 21. The primary difference is that a slightly larger threshold is needed for informed instruction prefetching to obtain the same results.

Since the difference in uninformed and informed instruction prefetching is whether distance on the Fetch Unit stack is calculated relative to the beginning versus the end of a function, in these benchmarks the typically executed functions are not long enough to greatly change the distance which contributes to ahead. For benchmarks with much larger functions, informed prefetching should prove more efficient, but the average fetch latency would be smaller, and thus the benefit from instruction prefetching would be less. For the benchmarks chosen, however, it is clear that the added complexity of informed instruction prefetching is not compensated by performance.

4.3.3 Incremental Instruction Prefetching

Figure 24 shows the effect of threshold on the IPC of the candidate benchmarks when incremental instruction prefetching is used. Since the nature of the way ahead is calculated is fundamentally different for incremental instruction prefetching, the shapes of the curves
are different than that for uninformed and informed instruction prefetching. Note, however, that the best and worst speedups remain almost identical for each benchmark. For *pppp* this is between 23% at a threshold of 1 and 30% at a threshold of 2048. The best geometric mean speedup is 9.4%, and is achieved at a threshold of 2048. Note that this is on par with uninformed and informed instruction prefetching, and so the simpler implementation of incremental instruction prefetching far outweighs a statistically insignificant improvement gained from the other methods for calculating ahead investigated here.

Figure 25 shows the fetch cycle recovery rates for incremental instruction prefetching. As for uninformed and informed instruction prefetching, these curves show a similarity to the speedup curves, except that *pppp* is much more sensitive to the efficiency of the prefetching algorithm than other benchmarks.

### 4.3.4 Alternate Configurations

Both protected mode instruction prefetching and use of a separate prefetch cache were compared against the original instruction prefetching configuration. An infinite threshold was used for both of these alternate configurations. For fairness, the prefetch cache was
Figure 24: IPC speedups for incremental instruction prefetching

Figure 25: Fetch cycle recovery rates for incremental instruction prefetching
created by dividing the original I-cache; half was used for the new I-cache and the other half for the prefetch cache. All other parameters were held constant. Figure 26 shows how these methods fared against each other for the instruction prefetching candidates. The first column represents perfect I-cache behavior, in which an I-cache request always results in a hit. The remaining columns represent, respectively, incremental instruction prefetching at a threshold of 2048, instruction prefetching in unprotected mode at an infinite threshold, protected mode instruction prefetching, and unprotected mode prefetching with a prefetch cache. The use of a prefetch cache in protected mode was also investigated, but the IPU's requests were indeed starved, and so the results showed a nearly zero speedup for all 6 candidate benchmarks.

When threshold is infinite, unprotected mode and protected mode prefetching fair well against each other. The use of a prefetch cache does not prove advantageous, however; not for a single candidate does the prefetch cache method prove superior to either of the other two methods. Protected mode wins out over unprotected mode prefetching for 5 of the 6 candidates, but this advantage is small compared to the extra complexity which accompanies protected mode prefetching. Note also that using an infinite threshold compares well against
a threshold of 2048 for incremental instruction prefetching, ever though 2048 yielded the best geometric mean for the candidates. The only case where a significant difference is seen is again for *fpcpp*.

### 4.3.5 Instruction Prefetching Summary

Since unprotected mode instruction prefetching without a prefetch cache is the simplest configuration, and since it produces results on par with protected mode prefetching and better than unprotected mode prefetching with a prefetch cache, unprotected mode prefetching was used for further experimentation. Also, since incremental instruction prefetching is the simplest method for estimating ahead and since it produces results nearly as good as the more complex uninformed and informed instruction prefetching methods, incremental instruction prefetching was chosen as the method for estimating ahead. In particular, the threshold of 2048 was used for further experimentation since it yielded the best geometric mean speedup over the 6 candidate benchmarks.

IPU statistics for these benchmarks are given in Table 3. All six candidates exhibit speedups, although the degree to which this is successful varies greatly. What the behaviors of all of these benchmarks have in common is the IPU activity, which only varies from 31.31% to 46.94%. So the IPU spends a significant amount of time requesting lines, as opposed to re-reading previously-seen functions due to resets or a graph-like call structure. This, then, raises the question of whether the IPU is requesting the correct lines. And in fact the prediction rate is closely correlated with the speedup. The prediction rates vary greatly, from 92.92% down to 21.10%, and the increased traffic is due more to completely unnecessary lines than to contention due to I-cache size, the exception being *fpcpp*. Finally, note that only *fpcpp* is significantly affected by the specific threshold value; the IPU is limited by the threshold much less often for the other benchmarks. From these results, it seems clear that a great improvement to this method would be a way of preventing prefetching of lines which are never used; the cache contention tends to be less of a problem, even when the threshold is only used 10% of the time.

83
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Inc. Traffic</th>
<th>Pred. Rate</th>
<th>Contention</th>
<th>R-Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpppp</td>
<td>53.02%</td>
<td>92.95%</td>
<td>57.32%</td>
<td>2.53%</td>
</tr>
<tr>
<td>gcc</td>
<td>23.53%</td>
<td>34.51%</td>
<td>28.78%</td>
<td>3.20%</td>
</tr>
<tr>
<td>go</td>
<td>211.66%</td>
<td>44.45%</td>
<td>29.39%</td>
<td>1.32%</td>
</tr>
<tr>
<td>m88k瑕疵</td>
<td>90.46%</td>
<td>48.96%</td>
<td>32.74%</td>
<td>3.38%</td>
</tr>
<tr>
<td>perl</td>
<td>406.78%</td>
<td>21.10%</td>
<td>21.79%</td>
<td>2.11%</td>
</tr>
<tr>
<td>vortex</td>
<td>83.54%</td>
<td>53.65%</td>
<td>37.89%</td>
<td>3.92%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>IPU Act.</th>
<th>No. Loaded</th>
<th>T-Freq.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpppp</td>
<td>37.16%</td>
<td>167002198</td>
<td>0.07%</td>
<td>40.28%</td>
</tr>
<tr>
<td>gcc</td>
<td>46.04%</td>
<td>56318887</td>
<td>0.05%</td>
<td>81.13%</td>
</tr>
<tr>
<td>go</td>
<td>34.32%</td>
<td>17557505</td>
<td>0.04%</td>
<td>73.71%</td>
</tr>
<tr>
<td>m88k瑕疵</td>
<td>31.31%</td>
<td>5487147</td>
<td>0.04%</td>
<td>86.49%</td>
</tr>
<tr>
<td>perl</td>
<td>40.40%</td>
<td>19077765</td>
<td>0.04%</td>
<td>80.25%</td>
</tr>
<tr>
<td>vortex</td>
<td>33.50%</td>
<td>30836626</td>
<td>0.03%</td>
<td>6.68%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inc. Traffic</td>
<td>The increase in main memory bus traffic</td>
</tr>
<tr>
<td>Pred. Rate</td>
<td>The prediction rate, calculated as the percentage of lines prefetched which are requested by the Fetch Unit before being displaced</td>
</tr>
<tr>
<td>Contention</td>
<td>The component of increased traffic due to cache contention rather than misprediction</td>
</tr>
<tr>
<td>R-Freq.</td>
<td>The frequency (on a cycle-by-cycle basis) of which a reset occurred</td>
</tr>
<tr>
<td>IPU Act.</td>
<td>The frequency of IPU activity (i.e., how often the IPU waited on requested cache lines)</td>
</tr>
<tr>
<td>No. Loaded</td>
<td>The number of cache lines requested by the Fetch Unit with the IPU disabled</td>
</tr>
<tr>
<td>T-Freq.</td>
<td>The frequency with which the threshold prevented a prefetch request</td>
</tr>
</tbody>
</table>

84
Figure 27: IPC speedups versus values of $K$ and $N$

### 4.4 Data Prefetching

In this experiment, data prefetching was evaluated using static values of $K$ and $N$, with instruction prefetching disabled. Values of $K$ ranged from 0 to 3, while values of $N$ ranged from 0 to 7. This experiment shows primarily the benefit from using values of $N$ greater than 1, which means prefetching more than 1 iteration ahead. Refer to the third claim in Chapter 1.

Figure 27 illustrates the geometric mean speedup across all the data prefetching candidate benchmarks, plotted versus $K$ and $N$.

Note first that for all values of $N$, the geometric mean increases as $K$ increases from 0 to 3. This indicates that the regular stride detection mechanism is doing its job; no prefetching is done unless a definite regular stride is detected. The best result, a geometric mean speedup of 2.9%, is achieved at a $K$ of 3 and a $N$ of 2. This is a relatively flat portion of the curve, however, and the second best result of 2.5% is seen at a $K$ of 3 and a $N$ of 3.

The low geometric means are due to a wide variance in the results of the individual candidates. While some benchmarks respond well to stride-based data prefetching, some do
not at all, having largely irregular access patterns or other problems. For example, while \texttt{compress} appears to be a regular benchmark, investigation proved that \texttt{compress} performs far more loads to access its encoded prefix hash table than in reading or writing from its internal buffers. The irregular access of the hash table dominates \texttt{compress}'s behavior.

Nevertheless, many of the floating point benchmarks are regular, and thus stride-based data prefetching works well. Figure 29 shows the candidates for which data prefetching speeds up execution. The benchmark \texttt{hydro2d} exhibits a speedup of 41.7%, but even this value is not the best a static approach can achieve for \texttt{hydro2d}. At a \texttt{N} of 5 and a \texttt{K} of 3, \texttt{hydro2d} exhibits a speedup of 48.5%. This shows the limitation of the static approach; with static values of \texttt{N} and \texttt{K}, the best mean results can be achieved, but each benchmark can perform better with customized values.

In summary, results vary widely for the different data prefetching candidates. Two candidates exhibit slowdowns of 5% or greater, six exhibit minimal slowdowns of less than 5%, two exhibit minimal speedups of less than 5%, and five exhibit speedups of 5% or greater. The results range from a 15.5% slowdown to a 41.7% speedup at the chosen fixed values of \texttt{N} and \texttt{K}.
4.5 Compiler Support for Data Prefetching

This section evaluates the effectiveness of the compiler algorithm for encoding prefetch distances into load instructions. Three bits were taken from the 16 bit load offset to encode one of eight values. The value 000 was reserved to indicate a regular load (i.e., without prefetching). The remaining values served as indices into a static table of seven values of N. These seven values were chosen by examining the values of N calculated by the algorithm described in Section 3.2.4. A histogram of these values for all eighteen benchmarks is given in Figure 30.

The seven values chosen were those that minimized the error function:

\[
\text{err} = \sum_{i=1}^{8} \text{count}(i) \times \min_{V_j \leq i} (i - V_j) \tag{34}
\]

In this equation, the \( V_j \) values represent the seven values plus the sentinel value zero.

For each potential value \( i \), the closest value of \( V_j \) less than or equal to \( i \) is picked, and this difference is weighted by the number of occurrences of \( i \). The seven values which minimize this error function are 1, 2, 3, 6, 9, 12, and 19. The value encoded into each load instruction, then, is the largest of those which is less than or equal to the desired value calculated by
Figure 30: A histogram of calculated values of $N$ for the SPEC95 benchmarks

The results of this experiment are divided as they were for the experiment utilizing static values of $K$ and $N$. Figure 31 shows the results for the candidates which exhibited a slowdown in the previous experiment. Both the static speedups and the encoded $N$ speedups are given. Note that in 7 of these 8 cases, using an encoded $N$ surpasses a static value, even the case of compress.

Figure 32 shows the results for the candidates which exhibited a speedup using a static value of $N$. Note that using an encoded value of $N$ squeezes a modest amount of additional speedup out of 6 of these 7 candidates. Since the table of $N$ values includes 2, which yielded the best results as a static value, using an encoded value of $N$ can only fail when the algorithm calculating $N$ does not return the best value for the circumstances. While it is clear that this is happening in 2 of the 15 cases, the geometric mean speedup does rise from 2.9% to 5.5%.
Figure 31: Candidates exhibiting data prefetching slowdowns for static N and K

Figure 32: Candidates exhibiting data prefetching speedups for static N and K

89
4.6 Comparison Against Additional Hardware

The experiments in this section evaluate the cost effectiveness of prefetching. Refer to the fifth chain in Chapter 1.

With the trend of increasing chip transistor density, the question is not whether to increase chip complexity but rather how the additional transistors should be used. For this reason, an experiment was conducted to compare the prefetching method against increasing the superscalarity of the machine. For the prefetching runs, both incremental instruction prefetching with a threshold of 2048 as well as data prefetching using 3 bits of $N$ encoding were allowed to run simultaneously. For the comparison, the effective superscalarity of the basic machine was doubled; the number of each type of functional unit was doubled, the sizes of all buffers were doubled, the number of allowed outstanding conditional branches was doubled, and the width of the bus to the I-cache was doubled as well. Without doubling the I-cache bus width, the Fetch Unit would still have only been able to receive 4 instructions at a time. Other than the I-cache bus, no aspect of the memory subsystem was changed.

Figure 33 shows the results of this experiment for the integer SPEC2005 benchmarks. These results look disheartening, since doubling the CPU hardware wins over prefetching for every benchmark. This is largely due to long data dependence chains in the integer benchmarks which clog the buffers and prevent some ILP from being exploited. For the floating point benchmarks, however, more of the ILP can be exploited with less hardware. Figure 34 shows the results for the floating point benchmarks. The difference is staggering; prefetching yields a higher IPC than doubling the CPU hardware for 6 of the 10 benchmarks, and the improvement is as significant as the reverse for the integer benchmarks. Note also that while hydro3d did not perform well in instruction prefetching, and fpppp did not perform well in data prefetching, each benchmark is allowed to perform well when both instruction prefetching and data prefetching are enabled. This indicates that the independence of the prefetching mechanisms does not cause them to compete for resources enough to cause memory system thrashing.

The geometric mean speedup for doubling the CPU hardware is 8.4%, while the geometric mean speedup for prefetching is 7.1%. While doubling the CPU hardware wins
Figure 33: Prefetching vs add. CPU hardware - int benchmarks

Figure 34: Prefetching vs add. CPU hardware - fp benchmarks

91
out, it does so requiring many times more transistors than the prefetching mechanisms. All in all, the prefetching mechanism proves to be more cost effective than doubling the CPU hardware. And as the amount of existing CPU hardware increases, this trend continues. For example, the base machine simulated is not fully 4-way superscalar. Another experiment was conducted which compared a fully 8-way superscalar processor against prefetching on a fully 4-way superscalar processor. The results are given in Figures 35 and 36. They look much the same as in Figures 33 and 34, but doubling the hardware on the fully 4-way superscalar processor yielded a geometric mean speedup of 6.5%, while the addition of prefetching hardware yielded a close geometric mean speedup of 6.3%.

4.7 Comparison against Reinman and Baer/Chen

In this section, the IPU and LTB are compared separately and jointly to the methods proposed by Reinman et al. [109] and Baer and Chen [10, 24], respectively. The results from these experiments will validate the first and fourth claims, as stated in Chapter 1.

The problem with Reinman’s method is that instruction prefetching based on branches is undermined by both spatial and temporal locality in the I-cache; it ignores the fact that
the instructions which will be needed next but which are not currently in the I-cache are likely to be those further along in the current function or off in another function which will be called soon. The IPU addresses this directly, with varying degrees of success. In order to compare the two, one change was made to the experimental framework; the branch target buffer needed to be changed to store unconditional branches as well as conditional ones. Reimann’s assumption of a multi-port branch target buffer was used, and except for the type of control flow, each method behaved identically. For fairness, Reimann’s method was tuned the same way as the IPU: by using a lookahead value of powers of two, and the Reimann method was tried both with and without the addition of the interprocedural control flow of the IPU.

The results for the integer benchmarks are given in Figure 37, while the results for the floating-point benchmarks are given in Figure 38. The \( H \) denotes the IPU speedup, while the \( R \) denotes the Reimann method with a lookahead value of 1 without interprocedural control flow. The method did not behave well at all; inside inner loops which fit in the I-cache, no useful work was done, while in more irregular code the mispredicted branches sabotaged the effectiveness of the method, causing resets whenever a misprediction was
detected. In less superscalar machines, this would not be as serious a problem, but in highly superscalar architectures a branch is predicted nearly every cycle, and a 90% prediction rate in integer code means a reset of the method every ten cycles. As the lookahead went up, more unnecessary instructions were prefetched, adding to the bandwidth problem, and performance dropped. This is why such a small lookahead value yielded the best results for the I-prefetching candidates. Even so, the high reset rate prevented the method from going much astray, and thus the results for the method are nearly flat.

Next, Baer and Chen's method was compared against the LTB. The essential problem with Baer and Chen's method is that a branch-based lookahead is limited again by the size of the branch target buffer and the accuracy of the branch predictor. Another problem with using a simple history-based branch predictor is that the end of an inner loop acts as a barrier to the lookahead PC, it will keep following the loop and not break out of it until the final loop test is actually executed, triggering a reset due to the misprediction. So a reset is absolutely required at the completion of all iterations of an innermost loop, and the lookahead must work to get ahead again to be useful for the loads which follow the loop. And finally, the scheme fails to mask effectively the hit-wait times introduced by non-blocking
caches. As loop execution times go down due to increased processor superscalarity and the load latencies go up, the ideal time to prefetch becomes more than one loop iteration ahead. Baer and Chen's scheme always prefetches for the next iteration. A large enough lookahead distance causes the lookahead PC to progress more than one iteration ahead, but the RPT state is updated by the actual load calculation, not the lookahead.

A different problem is a scaling problem: in a 4-way superscalar machine, the lookahead must attempt to move ahead as many as four instructions per cycle. This means checking up to four instructions for matches in the branch target buffer, as well as checking the same instructions for matches in the RPT. A change in design, perhaps matching a block of 4 instructions with aid from instruction placement by the compiler, could ease the latter. As such, the scaling problem was ignored for the purposes of experimentation; the branch target buffer and RPT were given as many ports as were necessary to check for matches on up to four instructions. Baer and Chen gave a heuristic for determining the optimal lookahead distance: between 1 and 1.5 times the memory latency. However, with superscalar architectures, a multi-level cache hierarchy, and non-blocking caches, this heuristic is no longer useful. Therefore, Baer and Chen's method was also evaluated by using powers of
two for the target lookahead distance, both with and without interprocedural control flow.

The results for the integer benchmarks are given in Figure 39, while the results for the floating-point benchmarks are given in Figure 40. The use of an L2 cache affected this quite a bit, as most of the floating point benchmarks were able to see a significant speedup with a small lookahead distance. The integer benchmarks, however, were plagued with increased bandwidth due to unused prefetches, and the larger the lookahead distance grew, the worse the problem became. For this reason, the best results for the data prefetching candidates came with a lookahead distance of just two, with interprocedural control flow. For comparison, the results of hydro2d at this configuration are 16.1%, and just slightly higher at 17.5% at its peak, compared with 43.6% by the LTB. Also, the difference between employing interprocedural control flow and not is not statistically significant, even for hydro2d.

A followup experiment tried different static values of N, and better results were achieved for a value of 2 for N, which shows that the long hit-wait times caused by the non-blocking caches coupled with decreased loop execution time has pushed the optimal lookahead distance more than one iteration ahead. Figure 41 compares Baer/Chen results with the
default (NJ) values of (1,0) to those obtained with (2,3) while keeping all other parameters constant. A significant improvement is seen for the data prefetching candidates in almost every case.

A better solution might be found to this problem than just using a static value of 2 for N; since the lookahead PC does cycle through each iteration, a counter might be kept of lookahead hits, and this counter could be used for determining the N for that load. The counter could then be decremented when the load address was calculated for that iteration. The primary problems with this idea are in increased contention for accessing the RPT and how to handle branch mispredictions, especially as prefetching will always be done for iterations which are never actually executed.

Finally, the combined IPU/LTB approach was compared to a combined Reiman/Baer and Chen approach. The latter two were combined by having the RPT triggered by predecoded instructions which were prefetched by Reiman’s method, rather than by associative lookup in the BTB and RPT. The results are given in Figure 42 and Figure 43, and they are unremarkable in the sense that combining the methods gives results comparable to what would be expected if they operated independently. The only interesting note is that while
Figure 41: Chen sensitivity to N and K
Reinman’s method worked best overall from a lookahead value of 1, and Baer and Chen’s method from a lookahead distance of 2, the combined method works best from a lookahead distance of 16. This is likely to be noise, however, since the geometric mean speedup curves are relatively flat at low lookahead values.

4.8 Sensitivity to Machine Parameters

Thus far, the IPU/LTB prefetching method has been tuned to yield the greatest mean speedup, and it has been compared against alternate hardware configurations. This section examines the relationship of the speedup yielded by IPU/LTB to machine parameters which are external to the prefetching hardware itself. From this it will be possible to determine how current trends in machine design will affect the effectiveness of the prefetching mechanism.

Each parameter studied is taken one at a time in the sections below, and all other parameters are held constant while it is varied. The experiment in Section 4.8.4 in particular will validate the sixth claim made in Chapter 1.
4.8.1 Memory Access Pipelining

The experiments conducted above used a memory model in which the main memory could handle a new request a minimum of 50 cycles after the most recent prior request was initiated. More and more memory systems are being designed which allow multiple outstanding requests to be processed at once. For example, a memory system pipelined two deep would allow for a second memory request to be initiated 25 cycles after the first. Each request would still incur a 50 cycle latency, but requests could be fulfilled every 25 cycles instead of only every 50 cycles, which effectively doubles the maximum memory bandwidth.

An experiment was conducted in which pipelining was simulated one deep up to six deep. In each case the main memory’s ready time, the minimum amount of time before the main memory can accept a new request, is calculated by the following:

\[
\text{ready time} = \frac{50}{\text{degree of pipelining}}
\]  

The results of this experiment are shown in Figure 44. They were rather undramatic, as the burst-mode transfer of main memory coupled with the long cache lines of the L2 (16 words) combined to undermine the effect of pipelining the memory. The reason is that a
Figure 44: Speedup versus degree of pipelining

transfer of 16 words required 4 transfers on a 128-bit bus. The first transfer completed after the incurred latency of 50 cycles, and the remaining three each completed transfer 5 cycles after the previous transfer. So 15 of the next 50 cycles were being put to work, and any degree of pipelining over 3 was not worthwhile. Pipelining is beneficial when the memory latency is otherwise not masked by other memory operations. For this reason, another experiment was conducted in which all parameters were the same except for the size of the lines in the L2 cache, which were cut in half. The results of this experiment are shown in Figure 45.

Only two benchmarks show significant benefit from prefetching in a pipelined memory system, hydrofil and tomcat. Both of these benchmarks are limited more by the power of the memory system than by the power of the functional units in the CPU. An increased memory bandwidth allows the prefetching mechanism to reduce unmasked memory latency. The prefetching mechanism achieves a 47.8% speedup for hydrofil without pipelining, for 8-word L2 cache lines, but a 72.1% speedup for pipelining five deep and beyond. And tomcat yields a 29.5% speedup without pipelining and a 42.3% speedup with a memory system pipelined five deep and beyond.
Aside from the effectiveness of the prefetching mechanism, most of the benchmarks themselves do not significantly benefit from a pipelined memory system. Only hydrofl experiences a speedup of greater than 5% from just a deeply pipelined memory; this speedup is 11.5%. The next greatest benefit is from swim, which experiences a 4.25% speedup. In the former case, the pipelined memory serves execution of the benchmark as well as prefetching, and in the latter case, the pipelined memory benefits execution of the benchmark, but the benefit of prefetching hovers between speedups of 16% and 18%.

4.8.2 Memory Latency

The trend towards higher memory latencies threatens to adversely affect the effectiveness of this or any other prefetching mechanism. The reason for this is that as memory latency increases, more and more latency can be masked, up to a balance point, in which the maximum amount of latency can be masked. All memory latency greater than the balance point is unmasked and therefore directly affects the execution time of the benchmark. By definition, at the balance point for the benchmark and machine configuration the prefetching mechanism is most effective in reducing execution time by masking memory latency with
useful work. Beyond the balance point, however, prefetching benefit dwindles towards zero. It is therefore imperative that a comprehensive study of a prefetching mechanism investigate where the balance point lies.

An experiment was conducted in which the same machine and memory configurations were used, but the latency to main memory was varied from 15 to 400 cycles. The results for the integer benchmarks are given in Figure 46, and the results for the floating point benchmarks are given in Figure 47. The results show that of the integer benchmarks, the increased memory latency sabotages prefetching operation in compress, but it affects the other integer benchmarks very little. In other words, the optimal point tends to be low, but the degradation in prefetching effectiveness tends to be very slight. For compress, the irregular access patterns are not detected, and so the LTB issues data prefetch requests which do not benefit the CPU but do greatly increase the memory traffic, which is more and more serious as the memory latency increases. For the floating point benchmarks, it is clear that it is the benchmarks which benefit from data prefetching which are sensitive to main memory latency. For example, fpppp receives its benefit from instruction prefetching, and even though the speedup afforded by the prefetching mechanism is significant, fpppp is not significantly sensitive to changes in the main memory latency across the range of 15 to 400 cycles. On the other hand, the benchmarks hydro3d and tomat0, which benefit greatly from data prefetching, are very sensitive to changes in main memory latency, and the benchmarks su2cor, swim, and turb3d, which receive a lesser benefit, are also sensitive to a lesser degree. Of these, the only balance point which is less than 400 cycles is that for hydro3d, at 90 cycles. At 400 cycles, hydro3d is still receiving a speedup of 40.0%, and this benefit is sloping off slowly. For tomat0, the benefit is 18.2% and still climbing. Overall these results are very encouraging. They show that, for the benchmarks which benefit from the prefetching mechanism, the benefit from prefetching is still good at a 400 cycle memory latency and beyond.

183
Figure 46: Speedup versus main memory latency - int benchmarks

Figure 47: Speedup versus main memory latency - fp benchmarks
4.8.3 Bus Width

An alternative to memory access pipelining for increasing main memory bandwidth is to widen the main memory bus. This method allows the L1 to be filled more quickly, but even with large L2 cache lines, doubling the main memory bus width cannot afford as much speedup as a two-deep pipelined memory, if requests to main memory are frequent enough to make pipelining worthwhile.

As experiment was conducted in which the benchmarks were simulated with the same machine and memory configurations, except for the bus width, which varied between 32 bits and 512 bits. Anything beyond 512 bits would be useless, since the L2 cache lines themselves are 512 bits. The results of the experiment are shown in Figure 48 for the integer benchmarks and Figure 49 for the floating point benchmarks. For the integer benchmarks, the results show that an increased main memory bus width tended to help the effectiveness of the prefetching mechanism, although the difference tended to be small. The most dramatic improvement for the integer benchmarks was in increasing the bus width from 64 to 128 bits. For the floating point benchmarks, the results are not as clear-cut. As the bus width is increased from 64 to 128 bits, the prefetching mechanism also tended to operate more effectively for the floating point benchmarks, but for the benchmarks grid, au68k, alps8u, and waved, a greater effectiveness was seen at a 32-bit bus. This is because the data demands on memory are not as significant as the bus width is increased, and for these benchmarks the prefetching mechanism was able to mask memory latency even when the memory bus bandwidth was greatly reduced. As the memory bus bandwidth increased, there was less memory traffic and thus less memory latency to mask by CPU activity.

4.8.4 Cache Size

The SPEC95 benchmarks were designed to test execution speeds on machines with smaller caches than the cache size used in the preceding experiments. For this reason, an experiment was conducted to evaluate how the prefetching mechanism functions with smaller caches, and also for larger caches. In the experiment, the L1 caches were scaled from 4 Kb to 128 Kb each, and the L2 cache was scaled in proportion (i.e., the size of the L2 remained...
Figure 48: Speedup versus main memory bus width - int benchmarks

Figure 49: Speedup versus main memory bus width - fp benchmarks
Figure 50: Speedup versus L1 cache size - int benchmarks

32 times the size of one of the L1 caches). The results of this experiment are shown in Figure 50 for the integer benchmarks and in Figure 51 for the floating point benchmarks. The cache size on the x-axis represents the size of each L1 cache.

As the size of the L2 cache grows larger than the benchmark's code size, the benefit from instruction prefetching drops off sharply. Then, as the size of the L1 I-cache grows larger than the benchmark's code size, the benefit from instruction prefetching drops to essentially zero. The best example of this is the benchmark fppp, which has a code size of 67 Kb. As the L1 I-cache increases from 32 Kb (15 on the log base 2 scale) to 64 Kb, the instruction prefetching benefit plummets from a 34.3% speedup to a 3.6% slowdown.

Likewise, as the L2 cache and the L1 D-cache each grow larger than the working set of the benchmark, the benefit from data prefetching drops off. There are a number of benchmarks, however, which peak for cache sizes in the range shown; these benchmarks most notably include hydrodd and tomcat. The reason for the peaking behavior is that at very small cache sizes, the prefetching mechanism and the load units thrash against each other more often. On the whole, however, both the integer and floating point benchmarks benefit even more from prefetching when smaller caches are employed. This indicates only
a small amount of thrashing, even for very small caches.

At the other end of the graph, the results would indicate that prefetching will become less worthwhile as cache size increases. Indeed this would be the case if cache demand did not also increase. As machines have been designed which are faster than their predecessors and endowed with larger, faster memories and caches, and with accompanying functional units to exploit available ILP, applications have arisen which increase the demands on the memory system, both from instruction fetches and data loads. Since the practical limiting factor on new applications is in fact the performance of these applications on state-of-the-art machines, the trends towards greater demand on the memory system should continue, and by extension, the need for prefetching to supplement what the memory systems alone can deliver.
CHAPTER V

CONCLUSIONS

This research has yielded new prefetch units and demonstrated several results from modeling and simulation of these new prefetch units and through comparison to prior designs. These results include:

1. Inter-procedural control flow is better suited to instruction prefetching than branch-based control flow in modern superscalar processors. This was shown through the comparison to the instruction prefetching research performed by Glenn Reiman et al. [109], in Section 4.7.

2. A simple incremental scheme to control instruction prefetching distance is more effective than protected-mode caches or separate prefetch caches, and it is nearly as effective as more elaborate schemes based on the call stack. This was shown through the comparison of various flavors of inter-procedural instruction prefetching in Section 4.3.

3. Modern memory system latencies require data prefetching more than one iteration ahead in inner loops. This was shown in the relative results of using static prefetching distances in Section 4.4.

4. Prefetch distance encoding performs better than lookahead mechanisms for stride-based data prefetching in modern superscalar processors, due to this need of larger distances. This was shown through the comparison to the lookahead-based data prefetching research performed by Jean-Loup Baer and Tien-Pu Chen [10, 24] in Section 4.7.

5. Prefetching is cost effective. This was shown through the comparison of adding prefetching hardware to adding more functional units, in Section 4.6.
6. The IPU/LTB approach holds up well against a trend of an increasing application/cache ratio. This was shown in the analysis of hardware trends in Section 4.8.
BENCHMARK CACHE BEHAVIOR

Each of the benchmarks was executed in full in order to obtain its execution time; then each was executed again, and statistics were collected at regular intervals so that 300 total samples would be collected for the benchmark. Three such statistics were plotted in order to determine fair execution ranges for each benchmark. The first is CPI, or the average number of cycles required to commit an instruction. The second is AFL, the average memory latency for fetch operations. The third is ARL, the average memory latency for read operations.

Some of the benchmarks exhibited flattening of these curves, indicating a steady-state of machine operation and cache behavior. Others exhibited periodic behavior; cache behavior would often get suddenly worse at the beginning of an outer loop iteration, and gradually better as the iteration progressed. For benchmarks exhibiting steady-state behavior, a range of 100 million committed instructions were selected, beginning at or just past the elbow of the flattening. For benchmarks exhibiting periodic behavior, a range was selected beginning after the completion of the first iteration, for a whole number of iterations. The ranges chosen are ranges of committed instructions, even though they were chosen from the cycle ranges to which they correspond on the graphs. This is so that the amount of work remains constant from configuration to configuration, and the IPC is calculated by the amount of time required to complete this fixed amount of work.

In the plots presented in this appendix, the execution ranges chosen are shown as a gray rectangle. In a few cases, the best chosen range was more than half of the total execution; in these cases the entire benchmark was used.
Figure 52: Cache behavior for benchmark applu

Figure 53: Cache behavior for benchmark apsi
Figure 54: Cache behavior for benchmark compass

Figure 55: Cache behavior for benchmark fppp
Figure 56: Cache behavior for benchmark gcc

Figure 57: Cache behavior for benchmark go
Figure 58: Cache behavior for benchmark hydro2d

Figure 59: Cache behavior for benchmark ljpeg
Figure 60: Cache behavior for benchmark 2.

Figure 61: Cache behavior for benchmark m88ksim.
Figure 62: Cache behavior for benchmark mgrid

Figure 63: Cache behavior for benchmark perl

Figure 64: Cache behavior for benchmark su2cor
Figure 65: Cache behavior for benchmark swim

Figure 66: Cache behavior for benchmark tomcatv

118
Figure 67: Cache behavior for benchmark turb3d

Figure 68: Cache behavior for benchmark vortex

119
Figure 69: Cache behavior for benchmark wave5
APPENDIX B

MIRRORED SRAM

Mirrored SRAM (mSRAM) is a necessary component of the return address stack in this prefetching scheme, since the IPU requires synchronizing its stack with that of the Fetch Unit on a reset. Conventional SRAM is composed of cells which each store 1-bit, and the bit value in each cell is only accessible for output when the input /SEL is asserted. (See Figure 7-50 in [133] for a logic diagram of an SRAM cell.)

This design can easily be extended to incorporate two bits labeled A and B which can operate independently, but which can be synchronized at the assertion of one of the inputs. Figure 70 is a logic diagram for an mSRAM cell. Both the bits A and B have the same inputs and output as the conventional SRAM cell. There is an additional input, however, labeled /DMP which, when asserted, causes the A bit to be dumped (written) to B.

An array of mSRAM cells may be arranged in the same way as SRAM cells, as shown in Figure 71. There are separate A and B data input lines (DINAn and DINBn), separate A and B address lines (AAn and ABn), separate A and B control lines (/WEA, /CSA, and /WEA, and /WEB, /CSE, and /WEB), and separate A and B data output lines (DOUTAn and DOUTBn). Additionally, the mSRAM in Figure 71 has an input /DMP, which directly feeds into each mSRAM cell. Due to fanout limitations, it is necessary to buffer /DMP at regular intervals; however, these buffers are not shown in the diagram.
Figure 70: Functional behavior of an mSRAM cell
Figure 71: Internal structure of an 8x4 mSRAM
APPENDIX C

IPU LOGIC DIAGRAM

In this section, logic diagrams are given to illustrate the function of the IPU, and to provide some insight as to the degree of its complexity and transistor requirements. All custom logic is defined in terms of gates or well-known logic devices, such as DC flip-flops and multiplexers. Where a single line represents more than one physical wire, the convention of an intersecting diagonal line with the number of wires is used. When no such key is given anywhere along a wire's paths and branches, the width is one. Intersections are noted explicitly with a black dot; crossed lines with no dots do not intersect.

Figure 72 gives the overall logic for the IPU. Since the design of the RAS (return address stack) is critical to the IPU, the RAS itself is included in this diagram. And since the RAS handles two logical stacks, one for the IPU and one for the Fetch Unit, the interface between the Fetch Unit and the RAS is also included in this diagram. The Fetch Unit inputs to the RAS include RAL_FU, the return address input, /PSILFU, the signal indicating whether to push RALFU onto the Fetch Unit's RAS, and /POP_FU, the signal indicating whether to pop the top return address from the Fetch Unit's RAS and send it back along the output RAO_FU. Since the RAS can overflow and only the bottom 256 entries are stored on the stack, a pop to the RAS may be for an item not stored. The output RAYFU is asserted when the return address given in RALFU is valid.

Two more inputs are required in order to reset the IPU. These are BST, which triggers a reset when asserted, and PC, the Fetch Unit's current PC. On a reset the RAS stacks are synchronized, and /PC is set to PC. The flow of logic works as follows:

1. When state is IDLE, a prefetch request is issued by asserting PF. The address is given on PA. These signals go to the I-cache.

2. When the requested cache is received, the input REOC is asserted, and CL contains the
Figure 72: Logic diagram for the IPU
contents of the cache line. Note that there is some hand-waving here with regard to cache line alignment. The actual address sent to the I-cache will be the address of the first byte on the cache line into which \( ipPC \) points. Likewise, when a cache line is received in CL, the first \( (ipPC + 4) \) modulo 8 words are of no interest, since \( ipPC \) points past them, and they should be shifted out and nops shifted in.

3. The decode module picks the first instruction with a change in procedural control flow. The output FC is asserted on a function call; FR is asserted on a function return; and INC is the index of the instruction encountered (0 for the first, 1 for the second, etc.). If no change in procedural control flow is found, neither FC nor FR is asserted, and INC is 8 (the number of instructions in the cache line). OFST is the encoded offset for a function call. In the DLX architecture, this is a sixteen-bit 2's complement value which represents target \(- PC - 4\). If FC is not asserted, OFST is undefined.

4. Function calls and function returns cause proper access in the IPU's RAS. The input DIN is the data input \( B \), which is the return address to push for the IPU. This value is accordingly calculated by the module labeled RA. This value would be at index INC+1 past the current ipPC, so the entire calculation is \( ipPC + 4(INC + 1) \).

5. The value of ahead is updated based on INC, the progress made by the IPU, and DPC, the progress made by the Fetch Unit. DPC is \( \Delta PC \), as discussed in Section 3.1.1.3. The module ahead produces the output ADK, which is asserted as long as ahead is less than threshold.

6. The value of ipPC is updated, depending on whether a function call or return or neither was encountered, and is set to PC on a reset.

7. state is updated, depending on whether a cache line was received, and whether the IPU can proceed with prefetching based on the value of ahead. This module also handles resets and attempting to follow invalid return addresses; more detail is given below.

126
C.1 Decoding Logic

The diagram for the IPU module labeled decode is shown in Figure 73. The flow for this module is fairly simple. The cache line labeled CL is split into its 4-bit 32-bit instructions. The diagram is meant to convey that the leftmost instruction represents the one with the lowest memory address. Each instruction passes through an instruction decoding module labeled insn decode. Each module produces three outputs: FR, which is true if the instruction is a function return, FC, which is true if the instruction is a function call, and a 16-bit output OFFSET, which is the encoded offset for the function call. FC and FR should be mutually exclusive. From there, an active-low signal is created which indicates a change in control flow (either FC or FR), and the 18 bits of output are bundled together for clarity in the diagram.
The active-low signals indicating procedural control flow are fed into a 74LS148 priority encoder, enabled by the RREG input. The priority encoder finds the first active input and returns the input number in /A2, /A1, and /A0. The output /OB is set low when an active input is found, and high when no active input is found. The encoder gives highest priority to input /I7, and lowest to input /I0. For this reason, the leftmost instruction is attached to /I7, so that lower memory addresses get priority.

The 18-bit decoded information from each instruction are then fed to a multiplexer, which picks the first with a change in control flow, as determined by the priority encoder. The 18-bit output from the multiplexer is shown debundled for clarity. The output to INC is set to be \(7 - (A2, A1, A0)\) if a change in control flow is found, and 8 if no change is found. This is the amount to increment \(\text{ip}PC\). As mentioned before, to account for cache alignment, the default value should not actually be 8 but \(8 - \left(\frac{\text{ip}PC}{4}\right) \mod 8\).

Figure 74 gives the logic diagram for the instruction decoding module. The instruction is fed to two sub-modules; one to detect function returns, and the other to detect function calls. The SuperDLX compiler uses the instruction \(jr\ r31\) on a function return. While \(jr\) is any kind of indirect jump, the compiler is restrained from using \(r31\) for anything but returns. The 6-bit opcode \(for\ jr\ in\ DLX\) is 101101, while the \(r31\) is encoded in the following 5 bits as 11111. Consequently, the function return detection logic tests the first 11 bits against 10101111111 and discards the remaining 21.

Similarly, the DLX function call instruction is \(jal\) with an opcode of 111100. The next 10 bits are not used for \(jal\), but the remaining 16 encode the offset for the destination function. Thus, the function call detection logic saves the 16-bit offset and tests the first 6 bits against 111100.

This is the most architecture-dependent portion of the IPU's logic, although the sizes of encoded offsets, entire instructions, and cache lines may be different for different architectures.
Figure 74: Logic diagram for the instruction decode module

C.2 Return Address Stack Logic

Figure 75 shows the logic diagram for the RAS. The logic is identical for the A and the B inputs; a 32-bit mSRAM register is used for the stack pointer SP, and a 256x32 mSRAM array is used for the stack proper, with the DMP input used to dump the contents of A into B. Refer to Appendix B for more information on the design of mSRAM. This particular implementation of the RAS stores up to 256 addresses but keeps a full 32-bit SP so that even when the stack overflows, when it shrinks back to 256 entries or less it can resume returning values. This also means that a valid output (VA and VB) is needed to indicate whether a popped value is valid (i.e., found in the mSRAM).

Since the logic is identical for the A and B inputs, only the logic for the set of A signals will be discussed here. Most of the logic is used for the inputs to the mSRAM; r write is enabled via WEA if a push is requested (PSHA) and SP is less than 256. The RAS is enabled via CSA on either PSHA or PGPA, and the output is enabled via OEA on PGPA. The data input DINA is the return address passed in via DIA, and the address AA depends on the function
requested; on PSHA it is SP; otherwise it is SP − 1. SP is changed on either PSHA or POPA; on PSHA it is incremented, and on POPA it is decremented. VA is asserted if a pop was requested and if the decremented value of SP, the one used as the address in the read, is less than 256.

C.3 ipPC Logic

Figure 76 gives the logic diagram for changes to ipPC. The DC flip-flop represents the stored value of ipPC, and the inputs do nothing more than inform the multiplexer at the bottom which value should be stored in ipPC. The DC multiplexer input is the current FC, and it is selected when RST is asserted. The D1 input is ipPC + INC × 4 + OFFSET + 4, which is the destination address of a function call, and so it is selected when FC (but not RST) is asserted. The D2 input is RA, the return address from the RAS, and this value is selected when FR and FV (but not RST) are asserted. Finally, the D3 input is ipPC + INC × 4, and this is selected when no procedural control flow is detected and the IPU is not reset. This input is also selected when a function return is seen but the value returned from the RAS is not valid. In such a case, the value of ipPC does not actually matter, since the IPU goes into WAIT mode (see Section C.4).

The remaining logic is used to produce the multiplexer’s selector switches S1 and S0. Great advantage is taken for simplicity for the two impossible input combinations: both FC and FR asserted, and RV asserted without FR.

C.4 State Logic

Figure 77 shows the logic diagram for the IPU’s state. The value of state can be IDLE, RECEIVING, or WAIT; in the implementation in Figure 77, this has been implemented by assigning a DC flip-flop to each of the three possible states. Only two are necessary, but three have been used for clarity. At any given time, one of the three flip-flops stores a 1, and the other two store 0’s. IDLE is represented by the topmost flip-flop holding the 1; RECEIVING is represented by the middle flip-flop holding the 1; and WAIT is represented by the bottom flip-flop holding the 1.

Actually, only the lower 8 bits are used in as the address. If SP is 256 or greater, the write will not occur. The read will occur, and a bogus address will be returned on D0, but VA will be false.

131
Figure 76: Logic diagram for the IPU's ipPC module
Figure 77: Logic diagram for the IPU's state module
The new state is calculated in three standard sum-of-products (SOP) logic circuits, and a fourth is used to indicate that instruction prefetching should occur. The first (leftmost) SOP circuit is used to indicate whether the new state will be IDLE. This is the case in either one of two circumstances: if the IPU is reset or a cache line is received in RECEIVING mode. Three products are needed instead of two in order to handle an invalid return address from the RAS, in which case state becomes WAIT.

The next SOP circuit is used to indicate whether the new state will be RECEIVING. This is the case if state is IDLE and ahead is less than threshold, in which case a prefetch will occur, and if the cache line is not also received in the same cycle. If the cache line is received, the state will remain IDLE so that another prefetch may take place during the next cycle.

The third SOP circuit is used to indicate whether the new state will be WAIT. This is the case if the IPU needs to return from a function but cannot since the RAS has overflowed, or during initialization. If none of the flip-flops holds the 1, as is the case during power-on, the state will be set to WAIT. Also, for both the RECEIVING and WAIT circuits, a reset via RST overrides the other logic and forces the lines low.

The fourth SOP circuit feeds the output PF, which indicates that a prefetch should occur during that cycle. It is true when state is IDLE and ahead is less than threshold.

If any of the three signals indicating the new state are asserted, it means that the state may change. So while each of the three signals feeds the D input of its respective DC flip-flop. The logical OR of the three signals feeds all three C inputs.
REFERENCES


135


136


137


138


139


140


145


Reid Harmon was born on February 16, 1972 and grew up in Anderson, S.C. He attended Clemson University from 1990-1994, where he obtained a Bachelor's of Science degree in Computer Engineering. Since then he has worked on his Ph.D. in Computer Science at Georgia Tech. In 1996 he married Angie Shoemake from Hartsville, S.C., and in 1998 he bought a house and moved back to S.C. Since that time his Ph.D. research has been conducted remotely. He is the Chief Technology Officer for a small company called inREACH, which provides consulting, custom programming, and solutions for the healthcare industry. His son Chase was born in 1999, and Matthew in 2003, just five days after the defense of his research.