A Stateless, Content-Directed Data Prefetching Mechanism

Robert Cooksey*Intel Verification Bevearton, OR
Stephen Jordan Intel Desktop Products Group Intel, Beverton, OR
Dirk Grunwald Dept. of Computer Science University of Colorado Boulder, CO

Abstract

Although central processor speeds continues to improve, improvements in overall system performance are increasingly hampered by memory latency, especially for pointer-intensive applications. To counter this loss of performance, numerous data and instruction prefetch mechanisms have been proposed. Recently, several proposals have posited a *memory-side* prefetcher; typically, these prefetchers involve a distinct processor that executes a program slice that would effectively prefetch data needed by the primary program. Alternative designs embody large state tables that learn the miss reference behavior of the processor and attempt to prefetch likely misses.

This paper proposes *Content-Directed Data Prefetching*, a data prefetching architecture that exploits the memory allocation used by operating systems and runtime systems to improve the performance of pointer-intensive applications constructed using modern language systems. This technique is modeled after conservative garbage collection, and prefetches "likely" virtual addresses observed in memory references. This novel prefetching mechanism uses the underlying data of the application, and provides an 11% speedup using *no additional processor state*. By adding less than $\frac{1}{2}$% space overhead to the data cache, performance can be further increased to 12.6% across a range of "real world" applications.

1 Introduction

In early processor designs, the performance of the processor and memory were comparable, but in the last 20 years their relative performances have steadily diverged [4], with the performance improvements of the memory system lagging those of the processor. Although both memory latency and bandwidth have not kept pace with processor speeds, bandwidth has increased faster than latency. Until recently, a combination of larger cache blocks and high bandwidth memory systems have maintained performance for applications with considerable spatial reference locality. A number of data prefetching methods have been developed for applications that have regular memory reference patterns. Most hardware prefetch mechanisms work by recording the history of load instruction usage, and index on either the address or the effective address of load instructions [3, 6, 11]. This requires the prefetcher to have observed the load instruction one or more times before an effective address can be predicted and can very work well for loads that follow an arithmetic progression.

*Portions of this work was performed while Bob Cooksey was a student at the University of Colorado

At the same time, modern managed runtime environments such as Java at ".NET", rely on dynamic memory allocation, leading to reduced spatial locality because of these "linked data structures". Although these applications do benefit from prefetchers designed for regular address patterns, their irregular access patterns still cause considerable delay. An alternative mechanism is to try to find a correlation between miss addresses and some other activity. The correlation [2] and Markov [5] prefetchers record patterns of miss addresses in an attempt to predict future misses, but this technique requires a large correlation table and a training phase for the prefetcher. Roth *et al.* introduced related techniques for capturing producer-consumer load pairs [12]. compiler-based techniques [9] insert prefetch instructions at sites where pointer dereferences are anticipated. Luk and Mowry [8] showed that a greedy approach to pointer prefetching can improve performance despite the increased memory system overhead. Lipasti *et al.* [7] developed heuristics that consider pointers passed as arguments on procedure calls and inserted prefetches at the call sites for the data referenced by the pointers. Ozawa *et al.* [10] classify loads whose data address come from a previous load as list accesses, and perform code motions to separate them from the instructions that use the data fetched by list accesses.

Since prefetch mechanisms target different classes of program references, they can be combined to yield a more effective total prefetching behavior; this was explored for the Markov prefetcher [5] and it was found that stride prefetchers improve the performance of the Markov prefetcher by filtering references with arithmetic progressions, leaving more table space for references to linked data structures.

This paper investigates a technique that predicts addresses in pointer-intensive applications using a hardware only technique with no built-in biases toward the layout of the recursive data structures being prefetched, and with the potential to run many instances ahead of the load currently being executed by the processor. The ability to "run ahead" of an application has be shown to be a requirement for pointer-intensive applications [12], which traditionally do not provide sufficient computational work for masking the prefetch latency. Some hybrid prefetch engines [13] do have the ability to run several instances ahead of the processor, but require *a priori* knowledge of the layout of the data structure, and in some cases, the traversal order of the structure.

The method we propose, *Content-directed data prefetching*, borrows techniques from conservative garbage collection [1], in that when data is demand-fetched from memory, each address-sized word of the data is examined for a "likely" address. Candidate addresses need to be translated from the virtual to the physical address space and then issued as prefetch requests. As prefetch requests return data from memory, *their* contents are also examined to retrieve subsequent candidates. There are a number of ways to identify "likely" addresses. Conservative garbage collection *must* identify all possible addresses, and occasionally errs in a way that classifies non-addresses as possible addresses, causing garbage to be retained. Our problem is simpler since prefetching is not necessary for correctness and only serves to improve performance.

The content-directed data prefetcher also takes advantage of the recursive construction of linked data structures. By monitoring both demand load traffic *and* the prefetch request traffic, the content prefetcher is able to recursively traverse linked structures. A common challenge found in data prefetching of most application types is finding sufficient computational work to mask the latency of the prefetcher requests. Chen and Baer [3] approached this problem using a look-ahead program counter. The recursive feature of the content prefetcher achieves the same effect, in that it allows the content prefetcher to run ahead of the program's execution, providing the needed time to mask the prefetch request latencies.

The content-directed data prefetcher is designed to assist applications used linked data structures.

Applications with regular or "stride" references do not commonly load and then follow addresses. However, almost all modern processors provide some sort of stride prefetcher. Some prefetchers, such as correlation prefetchers, can mimic a stride prefetcher when used in isolation, and using only these prefetchers may inflate their true impact. In this paper, we combine content-directed data prefetcher with a stride prefetcher to achieve a robust prefetcher and always compare performance to a baseline architecture that includes a stride prefetcher to clarify the contribution of the content-directed data prefetcher.

The main contributions of this paper are:

- Establishing the concept of content-directed data prefetching. This paper contributes a novel history-free prefetching methodology that can issue timely prefetches within the context of pointer-intensive applications. The scheme also overcomes the limitations of "context-based" prefetchers (e.g. Markov prefetchers), which require a training period. The content prefetcher described in this paper does not require such a training period, and has the ability to mask compulsory cache misses.

- Evaluation of an effective pointer recognition algorithm. This is a core design feature of the content prefetcher. Without an accurate means of distinguishing an address from any other random data, content-directed data prefetching would not yield performance increases.

- Introduction of a *reinforcement mechanism* that accurately guides the content prefetcher down active prefetch paths. This method affords the content prefetcher with a storage-efficient way of recording the current prefetch paths, providing the needed feedback required to continue uninterrupted down a recursive prefetch path.

The rest of this paper is organized as follows. The simulation framework used to examine the feasibility and practicality of the content prefetcher is presented in Section 2, followed by the prefetcher design and implementation in Section 3. A performance evaluation of the content prefetcher embodiment is given in Section 4. Section 5 provides a design and performance comparison between the content prefetcher and the Markov prefetcher [5], with Section 6 concluding this paper.

2 Simulation Methodology

Results provided in this paper were collected using simulation tools built on top of a μop-level IA32 architectural simulator that executes *Long Instruction Traces* (LIT). Despite the name, a LIT is not a trace but is actually a checkpoint of the processor state, including memory, that can be used to initialize an execution-based simulator. Included in the LIT is a list of *LIT injections* which are interrupts needed to simulate events like DMA. Since the LIT includes an entire snapshot of memory, this methodology allows the execution of both user and kernel instructions. To reduce simulation time, rather than running an entire application, multiple carefully chosen LITs are run for 30 million instructions and are averaged to represent an application.

2.1 Performance Simulator

The performance simulator is an execution-driven cycle-accurate simulator that models an out-of-order processor similar to the Intel® Pentium® 4 microprocessor. The performance simulator includes a

detailed memory subsystem that fully models busses and bus contention. The parameters for the processor configuration evaluated in this paper are given in Table 1. Such a configuration tries to

Processor	
Core Frequency	4000 MHz
Width	fetch 3, issue 3, retire 3
Misprediction Penalty	28 cycles
Buffer Sizes	reorder 128, store 32, load 48
Functional Units	integer 3, memory 2, floating point 1
Load-to-use Latencies	L1: 3 cycles, L2: 16 cycles
Branch Predictor	16K entry gshare
Data Prefetcher	Hardware stride prefetcher
Busses	
L2 throughput L2 queue size	1 cycle 128 entries
Bus bandwidth Bus latency Bus queue size	4.26 GBytes/sec - 133 MHz 8B quad pumped 460 processor cycles - 8 bus cycles thru chipset (240) - 55 nsec DRAM access time (220) 32 entries
Caches	
Trace Cache	12 Kμops, 8-way associative
ITLB	128 entry, 128-way associative
DTLB	64 entry, 4-way associative
DL1 Cache	32 Kbytes, 8-way associative
UL2 Cache	1 Mbytes, 8-way associative
Block Size	64 bytes
Page Size	4 Kbytes

Table 1: Performance model: 4-GHz system configuration.

approximate both the features and the performance of future processors. Included with the base configuration of the performance simulator is a stride-based hardware prefetcher. It is important to note that all the speedup results presented in this paper are relative to the model using a stride prefetcher. By using both an accurate performance model, and making sure the base model is complete in its use of standard performance enhancement components (*e.g.* a stride prefetcher), the speedups presented in this paper are both accurate and realistic improvements over existing systems.

2.2 Avoiding Cold Start Effects

When executing the benchmarks within the simulators, it is necessary to allow the workloads to run for a period of time to *warm-up* the simulator. In the framework of this paper, it is important to allow the memory system to warmup to minimize the effects of compulsory misses on the performance

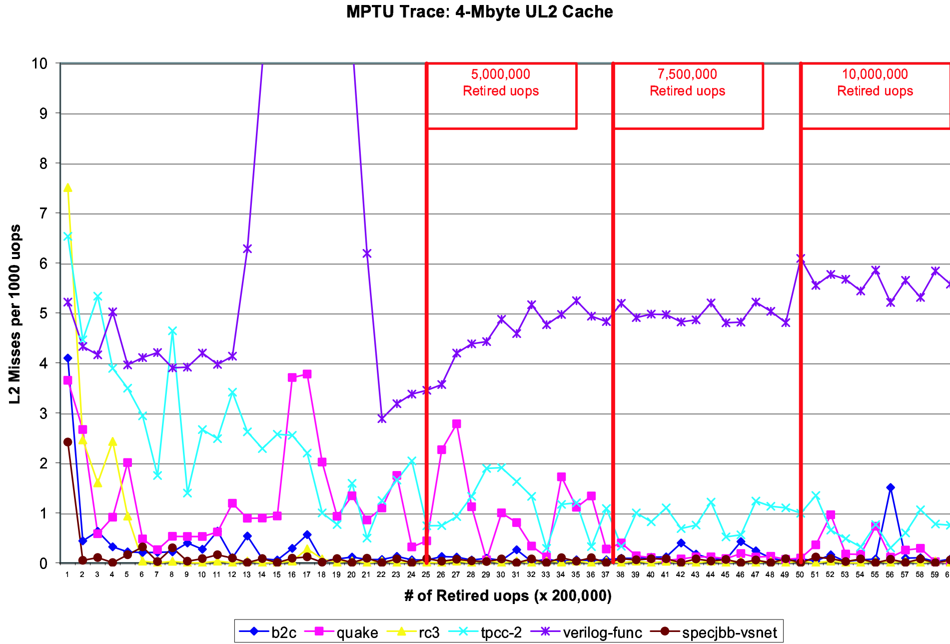

Figure 1: Non-cumulative MPTU trace for a 4-MByte UL2 Cache.

measurements. The conventional means for warming up a simulator is to allow the simulator to execute for a given number of instructions before collecting statistics. The metric used in this paper to establish this instruction count is *Misses Per 1000 μops* (MPTU). This is the average number of demand data fetches that will miss during the execution of 1000 μops. MPTU takes into account the number of operations being executed by the program, as well as the number of cache accesses, and provides a measure of a program's demand on the particular cache level. Figure 1 provides an MPTU trace of the second-level cache. The trace was generated for a processor configuration utilizing a 4-MByte *unified second-level cache* (UL2), as this will require a longer warm-up period. Using such a large cache guarantees that the instruction count will be valid for the UL2 cache sizes used during this study. The X-axis of the trace represents time, which is measured in retired μops. It should be noted that for readability purposes, the sequence has been limited to showing only one benchmark from each of the six workload suites (see Table 2). The MPTU trace shows a very distinct transient period from zero to 5,000,000 retired μops. By the time the execution of the programs reach 7,500,000 retired μops, the MPTU has reached a steady-state. Several spikes are seen later in the program's execution, but this is to be expected as both transitions between contexts within the program are encountered, and capacity misses start to appear. Using the trace as a guide, each application is allowed to execute for 5,000,000 IA32 instructions ($\approx 7,500,000\mu$ops), before the collection of performance statistics starts.

2.3 Workloads

The workloads used in this paper are presented in Table 2. The applications are either commercial applications or compiled using highly optimizing commercial compilers. The table lists the number of IA32 instruction executed, the number of μops executed, and the MPTU for both 1-MByte and 4-MByte second-level cache configurations. The workloads include applications from some of the most

Suite	Benchmark	Instructions	μops	L2 MPTU	
				(1 MB)	(4 MB)
Internet	b2b	30,000,000	40,369,085	1.04	0.83
Internet	b2c	30,000,000	49,979,480	0.13	0.13
Multimedia	quake	30,000,000	45,208,724	1.41	0.30
Productivity	speech	30,000,000	43,825,381	1.19	0.44
Productivity	rc3	30,000,000	47,226,941	0.43	0.33
Productivity	creation	30,000,000	52,939,628	0.56	0.24
Server	tpcc-1	30,000,000	52,944,514	1.88	0.68
Server	tpcc-2	30,000,000	53,149,114	2.29	0.87
Server	tpcc-3	30,000,000	51,719,199	2.49	0.87
Server	tpcc-4	30,000,000	51,896,535	2.05	0.70
Workstation	verilog-func	30,000,000	45,893,143	7.60	5.49
Workstation	verilog-gate	30,000,000	36,933,152	24.12	19.74
Workstation	proE	30,000,000	43,863,049	0.26	0.23
Workstation	slsb	30,000,000	49,697,532	3.23	2.74
Runtime	specjbb-vsnet	30,000,000	45,694,508	1.23	1.10

Table 2: Instructions executed, μop executed, and L2 MPTU statistics for the benchmark sets.

common productivity application types. This includes internet business applications (*e.g.* b2b), game-playing and multimedia applications (*e.g.* quake), accessibility applications (*e.g.* speech recognition), on-line transaction processing (*e.g.* tpcc), computer-aided design (*e.g.* verilog), and Java (or runtime) applications (*e.g.* specjbb).

3 Prefetcher Scheme and Implementation

This section presents the basic operation of the content prefetcher, and discusses some of the issues involved in the design decisions.

3.1 Basic Design Concept

The primary role of any prefetcher is to predict future memory accesses. The content prefetcher attempts to predict future memory accesses by monitoring the memory traffic at a given level in the memory hierarchy, looking expressly for virtual addresses (pointers). The prefetcher is based on the premise that if a pointer is loaded from memory, there is a strong likelihood that the address will be used as the load address (effective address) of a future load. Specifically, the content prefetcher works by examining the fill contents of demand memory requests that have missed at some level in the memory hierarchy (*e.g.* L2 cache). When the fill request returns, a copy of the cache line is passed to the content prefetcher, and the cache line is scanned for likely virtual addresses. If a candidate address is found, a prefetch request is issued for that address. The inherent difficulty in this prediction technique is trying to discern a virtual address from both data values and random bit patterns.

3.2 On-chip versus Off-chip

The content prefetcher can be implemented as being either on-chip, as an integrated piece of the processor's memory hierarchy, or as part of the memory controller (off-chip). Placing the prefetcher on-chip provides several benefits. First, with the content prefetcher residing at and prefetching into the L2 cache, the cache itself can be used to provide useful feedback to the content prefetcher. This can identify which prefetches suppressed a demand request cache miss, and whether the content prefetcher is issuing prefetches for cache lines that already reside in the L2 cache. Second, having the prefetcher on-chip provides the prefetcher with access to the memory arbiters, allowing the prefetcher to determine whether a matching memory request is currently in-flight. The final and the most compelling benefit is that placing the content prefetcher on-chip resolves issues concerning address translation. The candidate addresses being generated by the prefetcher are virtual addresses. However such addresses need to be translated to access main memory. With the content prefetcher on-chip, the prefetcher can utilize the on-chip data *translation look-aside buffer* (TLB). This is very important since statistics collected during simulation runs showed that on average, over a third of the prefetch requests issued required an address translation not present in the data TLB at the time of the request. The main drawback of an on-chip content prefetcher is that it may keep the pointer chasing critical path thru memory untouched. However, since the pointer chasing is now decoupled from the instruction stream execution, this critical time is now no longer additive to other limiting factors like computational work and branch mispredicts. Stated differently, pointer-intensive applications do not strictly utilize recursive pointer paths (*e.g* hash tables).

The major benefit of having the prefetcher off-chip is the possible reduction in the prefetch latency, as prefetch requests will not have to endure a full L2 to main memory request cycle; the pointer chasing critical path would be broken. An off-chip prefetcher could potentially maintain a TLB to provide the address translations. The problem with this implementation is that while this secondary TLB may be able to monitor the physical pages being accessed, it does not have direct access to the virtual addresses associated with the physical pages. Adding pins to provide the information could be an option. A second drawback to placing the prefetcher off-chip is the lack of feedback available to the prefetcher, as it is not receiving any direct reinforcement concerning the accuracy of the prefetches being issued.

After analyzing the benefits of both placement options, the decision was made to place the content prefetcher on-chip. The main reason is that in the applications focused on in this study, limiting factors other than true pointer chasing are mostly predominant. This will be confimed by the fact that 72% of the prefetches issued have their latency completely hidden. It will also be shown later in this paper that allowing the prefetcher to use feedback from the caches is extremely desirable for performance, and placing the content prefetcher on-chip affords the prefetcher its best access to the various feedback sources.

3.3 Address Prediction Heuristics

Some method must be designed to determine if a cache line value is a potential virtual address, and not a data value. The *virtual address matching* predictor originates from the idea that the base address of a data structure is hinted at via the load of any member of the data structure. In other words, most virtual data addresses tend to share common high-order bits. This is an common design in modern operating systems that is exploited by the content-designed prefetcher.

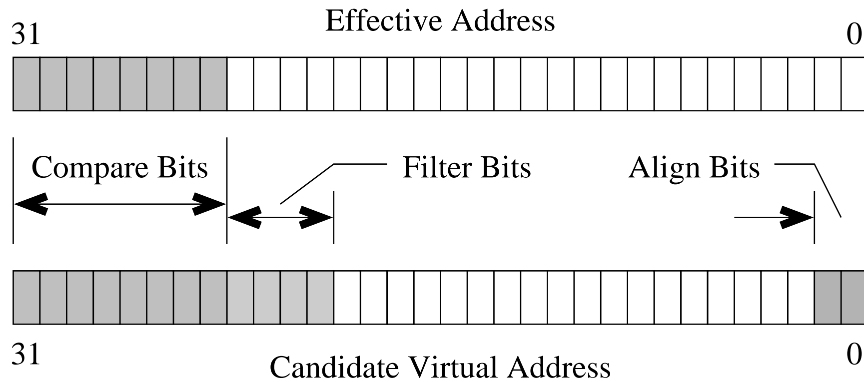

Figure 2: Position of the *Virtual Address Matching* compare, filter, and align bits.

More accurately, we assume all data values found within the structure that share a base address can be interpreted as pointers to other members (nodes) of the same structure. For this method, the effective address of the memory request that triggered the cache miss and subsequent fill is compared against each address-sized data value in the newly filled cache line. A given bit arrangement can be interpreted as a pointer, a value, or some random bits. Examples of the latter are compressed data or values taken out of their original byte grouping. A bit array is deemed to be a candidate address if the upper `N` *compare bits* of both the effective address of the triggering request and the bit arrangement match (see Figure 2), as this is a strong indicator that they share the same base address. The number of bits compared needs to be large enough to minimize false predictions on random patterns.

The virtual address matching heuristic works well except for the two regions defined by the upper `N` bits being all 0's or all 1's. For the lower region, any value less than 32-`N` bits will match an effective address whose upper `N` bits are all zeros. The same is true for the upper region, where a large range of negative values can potentially be predicted as a likely virtual address. This is problematic because many operating systems allocate stack or heap data in those locations.

Instead of not predicting in these regions, an additional filter as shown in Figure 2 is used to distinguish between addresses and data values. In the case of the lower region, if the upper `N` bits of both the effective address and the data value being evaluated are all zero, the next `M` bits, called *filter bits*, past the upper `N` compare bits of the data value are examined. If a non-zero bit is found within the filter bit range, the data value is deemed to be a likely address. The same mechanism is used for the upper range, except a non-one bit is looked for in the filter bit range. Using zero filter bits would mean no prediction within both extreme regions, while increasing the number of filter bits relaxes the requirements for a candidate address.

One further method of isolating addresses from data is using memory alignment. For memory efficiency reasons, most IA32 compilers will attempt to place variables and structures on 4-byte boundaries. Given the following data structure:

```
struct x {
    char a;
    struct x *next;
}
```

the compiler may not only place the base address of the structure on a four byte boundary, but it may

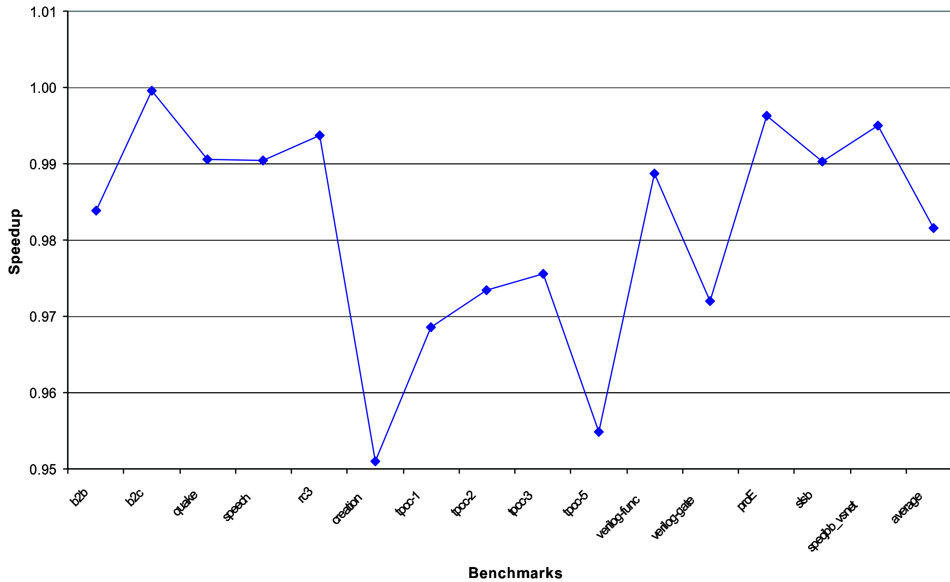

Figure 3: A limit study on the impact of excessive prefetching with zero percent accuracy.

also pad the size of the `char` to be 4-bytes so that both members of the structure will also be on 4-byte boundaries. The result is that all addresses related to the data structure will be 4-byte aligned. This alignment information can be used to the address predictor's benefit in two ways. First, with all the pointers being 4-byte aligned, the two least significant bits of each address will be zero. Therefore any candidate with a non-zero bit in either of these two bit locations can be discarded as a possible address. Figure 2 shows the position of the *align bits*.

The second benefit involves the scanning of the cache line. In a 64-byte cache line, if the predictor were to take single byte steps through the cache line, 61 data values would have to be scrutinized. Knowing that addresses only reside on 4-byte boundaries can allow the predictor to take a scan step of four through the cache line, reducing the number of data values that need to be examined from 61 to 16. This reduces the amount of work required during a cache line scan, and minimizes false predictions on non-aligned data values.

3.4 Content Prefetcher Design Considerations

3.4.1 Zero Accuracy Limit Study

The base design of the content prefetcher has the prefetch requests being filled directly into the second-level (UL2) cache. To quantify the impact of this prefetch fill policy, a dummy prefetcher that consumes 50% of the idle bus cycles with useless prefetch requests was added to the performance simulator. This approximates a prefetcher scheme flooding the memory system with useless prefetches, therefore polluting the second level cache, and giving an indication of the perturbation caused by a very low accuracy prefetcher.

The impact on the execution time of the benchmarks can be seen in Figure 3. As shown, allowing the prefetcher to run nearly unbounded with zero percent accuracy may result in a $\approx 3\%$pperformance slowdown. This highlights the need to maintain a reasonable accuracy with any prefetcher that directly

fills into the cache.

3.4.2 Recursion and Prefetch Priorities

The content prefetcher contains a recurrence component. Traditional prefetchers (*e.g.* stride, stream, correlating) observe only the demand fetch reference stream, or the demand miss reference stream subset, when trying to generate prefetch load addresses. The content prefetcher differs in that it not only examines the demand reference stream, but it also examines the *prefetch reference stream*. The result is that the content-based prefetcher will generate new prefetches based on previous prefetches (a recurrence relationship). This recurrence feature allows the prefetcher to follow the recursive path implicit in linked data structures.

The *request depth* is a measure of the level of recurrence of a specific memory request. A demand fetch is assigned a request level of zero. A prefetch resulting from a demand fetch miss is assigned a request depth of one. A *chained prefetch*, a prefetch resulting from a prior prefetch, is assigned a request depth equal to one more than the prefetch that triggered the recurrent prefetch request. The request depth can be seen as the minimal number of links since a non-speculative request.

This depth element provides a means for assigning a priority to each memory request, with this priority being used during memory bus arbitration. To limit speculation, prefetch requests with a depth greater than a defined depth threshold are dropped, and not sent to the memory system. An example of this recurrence mechanism is shown in the first part (left side) of Figure 4. A very small mount of space is allocated (2-4 bits, depending on the design) in the cache line to maintain the depth of a reference. That state is updated on a demand fetch or on a fill. This is the primary state used by the content-directed data prefetcher, and when using the final configuration of two bits per 64-byte cache block, amounts to less than a $\frac{1}{2}\%$ space overhead.

3.4.3 Feedback-Directed Prefetch Chain Reinforcement

A prefetch chain defines a recursive path that is being traversed by the content prefetcher. Any termination of a prefetch chain can lead to lost prefetch opportunities, as *no explicit history of the prefetch chain is stored*. To re-establish a prefetch chain, a cache miss must occur along the traversal path; the content prefetcher then evaluates the cache miss traffic and may once again establish the recursive prefetch chain. This is shown in Figure 5(a) where the base scheme results in a miss every four requests when the threshold depth has been set to 3.

To avoid this interruption in the prefetch chain, the content prefetcher includes a reinforcement mechanism that is based on the observation that *the chain itself is implicitly stored in the cache*. Thus any demand hit on a prefetched cache line provides the content prefetcher with the needed feedback required to perpetuate a prefetch chain.

The request depth value assigned a prefetch and cache line is not fixed for the lifetime of the prefetch memory transaction or cache line. When any memory request hits in the cache, and has a request depth less than the stored request depth in the matching cache line, it is assumed that the cache line is a member of an active prefetch chain. In the event that the fetch request depth is found to be less than the store request depth, two actions are taken. First, the stored request depth of the prefetched cache line is updated (promoted) for having suppressed the cache miss of the fetch request. This is consistent with maintaining the request depth as the number of links since a non speculative request.

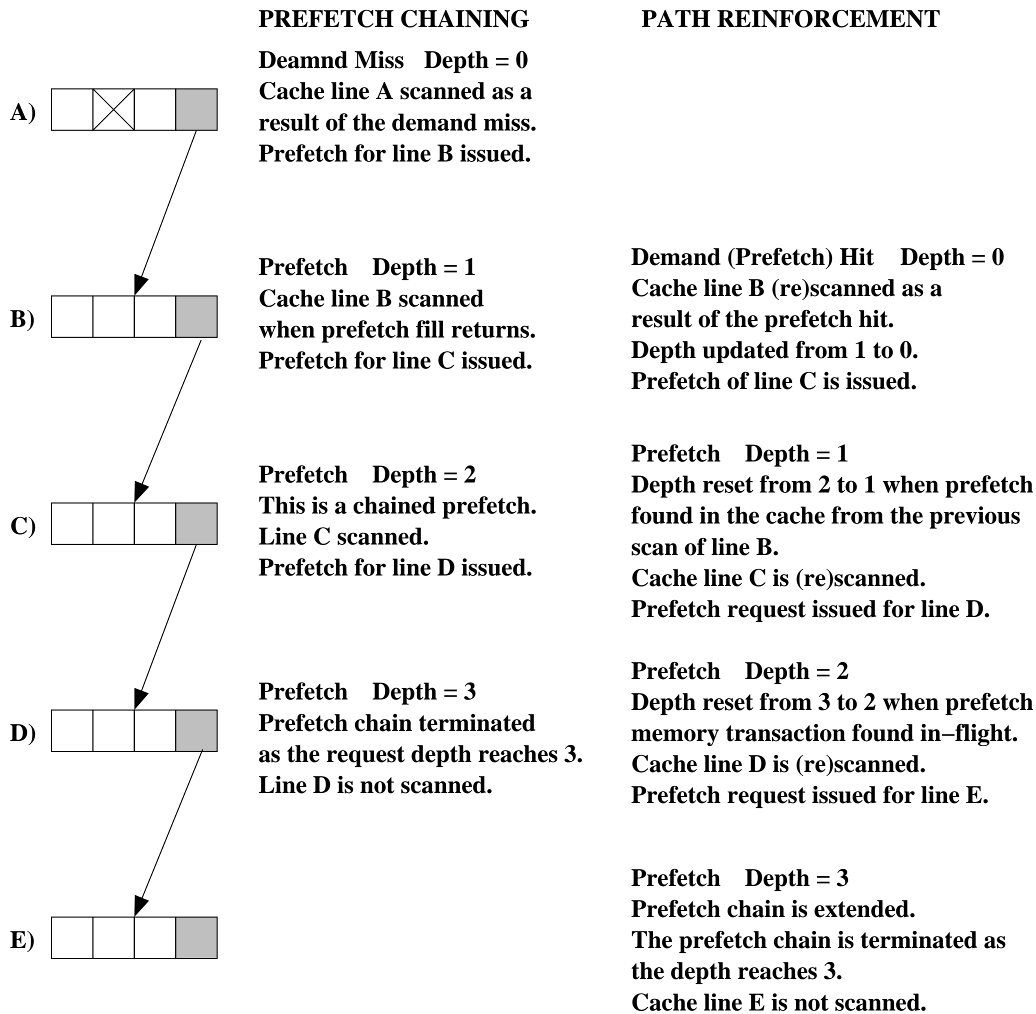

Figure 4: Example of *prefetch chaining* and *path reinforcement.*

The second action is that the cache line is scanned (using the previously described mechanism) in order to extend the prefetch chain.

So while a prefetch chain may be temporarily halted due to a large request depth, the ability to both change a cache line's stored request depth and rescan the cache line allows the content prefetcher to re-establish a prefetch chain. This reinforcement mechanism is applied to the example of the previous section in Figure 4 (right side). This mechanism also allows a prefetch chain to be restarted due to a prefetch request being dropped due to other unspecified reasons (*e.g.* insufficient bus arbiter buffer space).

Figure 5(b) graphically shows that a hit on the first prefetch request causes a rescan of the next two lines, resulting in the prefetch of the fourth line. This outcome happens for every subsequent hit, resulting in no other misses than the original one. In short, the reinforcement mechanism strives to keep the prefetching a number of depth threshold links ahead of the non speculative stream. This is a very effective way of controlling speculation without impairing potential performance improvement. The cost is a few more bits in each cache line as well as consuming second level cache cycles to re-

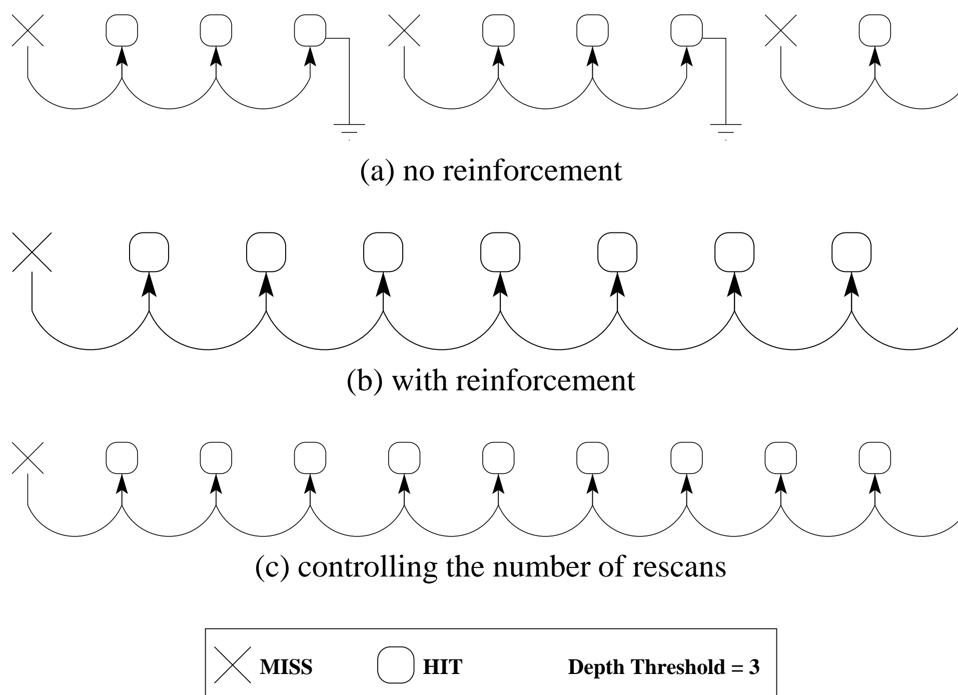

Figure 5: Re-establishing a terminated prefetch chain.

establish prefetch chains. Figure 5(c) shows how to half the number of rescans by re-establishing a chain only when the incoming depth is at least two fewer than the stored depth.

3.4.4 Bandwidth Efficiency: Deeper versus Wider

The recursive component of the content prefetcher can lead to highly speculative prefetches. One could take the stance that prefetches "deeper" in the prefetch chain are more speculative than those earlier in the chain. The prefetcher needs to place a threshold on this depth to limit highly speculative prefetches. But the question then arises how deep is deep enough, and can the memory bandwidth being allocated to highly speculative prefetches be better utilized elsewhere.

Until now, there has been an implicit assumption that the size of a node instance within a linked data structure is correlated in some manner to the size of the cache lines – obviously no such correlation exists since allocated data structures may span multiple cache lines. This means that no guarantees can be made about the locations of the data structure members within the cache lines. More specifically, the location of the pointer that leads to the next node instance of the recursive data structure may be in the same or the next cache line. Without these *next* pointers, the prefetch opportunities provided the content prefetcher will be limited.

To overcome this possible limitation, the content prefetcher has the ability to fetch *wider* references; that is, instead of speculatively prefetching down longer, deeper prefetch chains, the content prefetcher limits the depth of prefetch chains, and will attempt to prefetch more cache blocks that may hold more of the node instance associated with a candidate address. This is easily done by simply issuing one or more prefetches for the cache lines sequentially following the candidate address

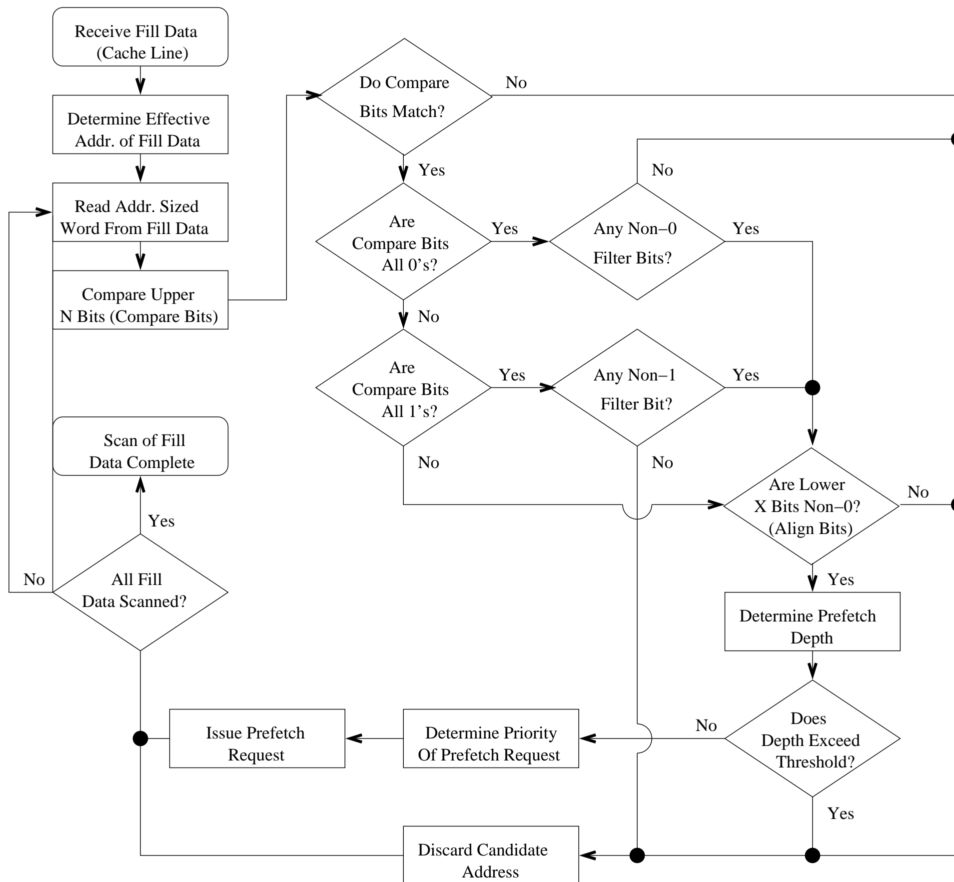

Figure 6: Flow diagram of the content prefetcher prediction mechanism.

as "next-line" prefetches. The right combination of depth versus width is established in Section 4.2, where empirical measurements are used to choose a specific configuration.

3.5 The Content Prefetcher Micro-Architecture

Figure 6 provides a flow diagram showing the operation of the content prefetcher prediction mechanism that makes use of the virtual address matching heuristic. As shown in the flow diagram, for an address-sized word to be validated as a candidate virtual address, it must meet the requirements defined by the compare, filter, and align bits, as well as the prefetch depth threshold. An examination of the flow diagram can lead to the conclusion that the scanning of a cache line for candidate addresses is strictly a sequential operation. This is not true, as such scanning is parallel by nature, with each address-sized word could be evaluated concurrently. Such a design can (and does) lead to multiple prefetches being generated per cycle.

The microarchitecture of the memory system that includes the on-chip content prefetcher is shown in Figure 7.

The content data prefetcher is located at the second cache level of the memory hierarchy, which is the lowest on-chip level. This provides the prefetcher with access to various components that can be used to provide feedback to the prefetcher. This memory system features a virtually indexed L1

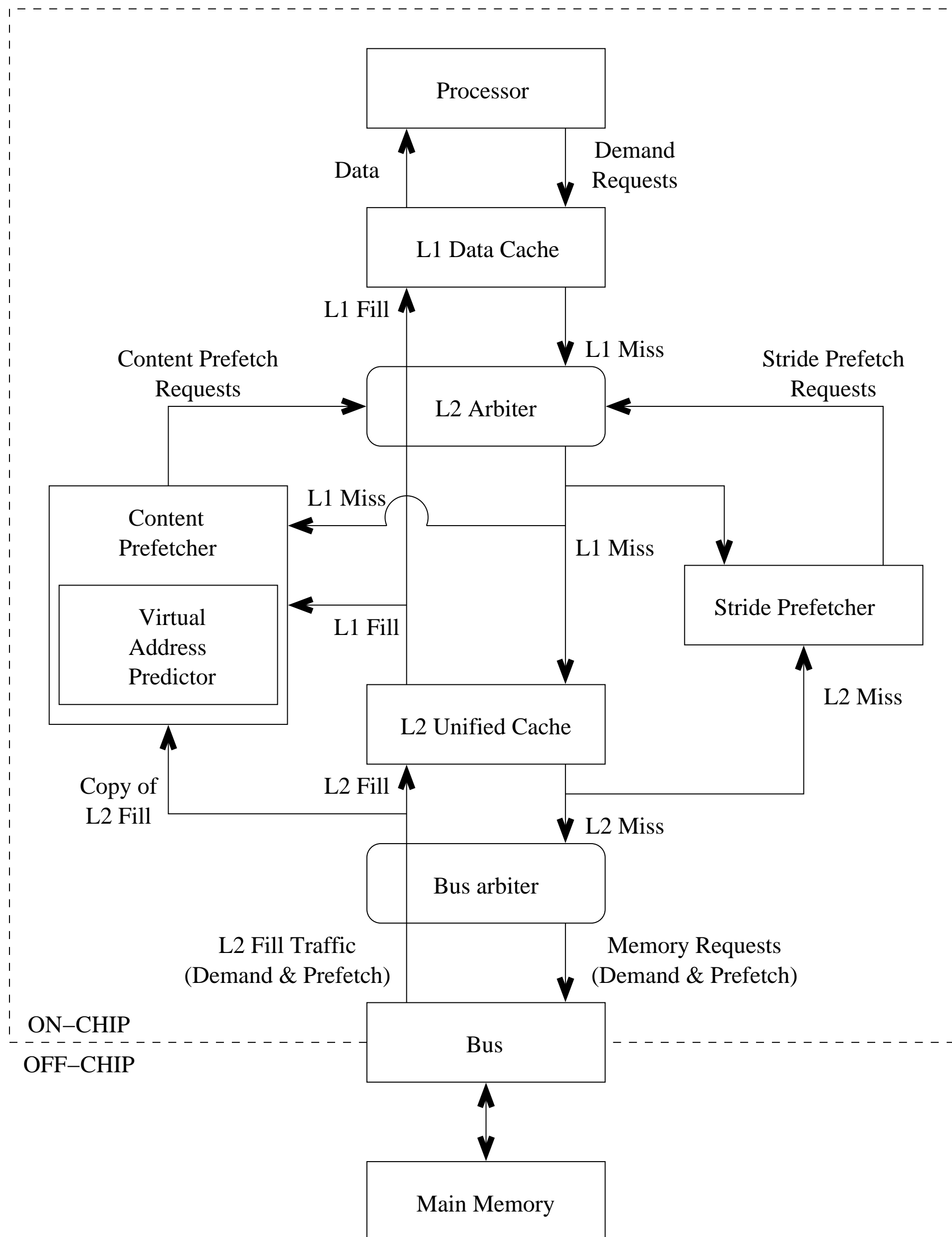

Figure 7: Microarchitecture with a memory system that includes both a stride and content prefetcher.

data cache and a physically indexed L2 unified cache; meaning L1 cache misses require a virtual-to-physical address translation before accessing the L2 cache. The stride prefetcher monitors all the L1 cache miss traffic and issues requests to the L2 arbiter. A copy of all the L2 fill traffic is passed to the content prefetcher for evaluation, with candidate addresses issued to the L2 arbiter.

Unlike many RISC processors, the processor model used in this study uses a hardware TLB "page-walk", which accesses page table structures in memory to fill TLB misses. All such "page-walk" traffic bypasses the prefetcher because some of the page tables are large tables of pointers to lower level page entries. Scanning such a cache line (which is full of virtual addresses) would lead to a combinational explosion of highly speculative prefetches.

The L2 and bus arbiters maintain a strict, priority-based ordering of requests. Demand requests are given the highest priority, while stride prefetcher requests are favored over content prefetcher requests because of their higher accuracy.

Having the content prefetcher on-chip provides the prefetcher with ready access to all arbiters. Before any prefetch request is enqueued to the memory system, both L2 and bus arbiters are checked to see if a matching memory transaction is currently in-flight. If such a transaction is found, the prefetch request is dropped. In the event that a demand load encounters an in-flight prefetch memory transaction for the same cache line address, the prefetch request is promoted to the priority and depth of the demand request, thus providing positive reinforcement (feedback) to the content prefetcher and insuring timely prefetches.

The arbiters are a fixed size. If in the process of trying to enqueue a request the arbiter is found to not have any available buffer space, the prefetch request is squashed. No attempt is made to store the request until buffer space becomes available in the arbiter. The behavior of the arbiters is such that no demand request will be stalled due to lack of buffer space if one or more prefetch requests currently reside in the arbiter. In this event, the arbiter will dequeue a prefetch request in favor of the demand request. The prefetch request with the lowest priority is removed from the arbiter, with the demand request taking its place in the arbiter. At this point the prefetch request is dropped.

4 Evaluation of the Content Prefetcher Implementation

This section begins by describing the steps taken to tune the address prediction heuristics. The tuning takes an empirical approach to adjusting the parameters specific to the address prediction mechanism. The results of the tuning are carried over to the cycle-accurate performance model, where they are used to demonstrate the performance improvements made possible when a content prefetcher is added to a memory system that already uses a stride prefetcher.

4.1 Tuning The Virtual Address Matching Predictor

Traditionally *coverage* and *accuracy* (shown in Equations 1 and 2) have been used to measure the *goodness* of a prefetch mechanism.

$$\text{coverage} = \frac{\text{prefetch hits}}{\text{misses without prefetching}} \tag{1}$$

$$\text{accuracy} = \frac{\text{useful prefetches}}{\text{number of prefetches generated}} \tag{2}$$

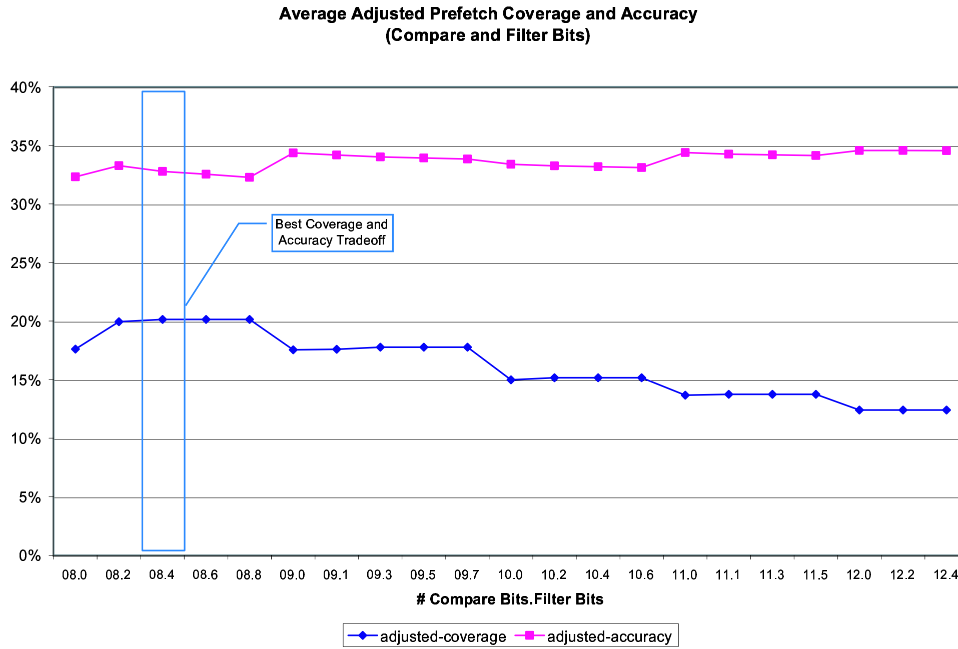

Figure 8: A summary of various compare and filter bit combinations. The horizontal axis shows a specific configuration of compare and filter bits. For example 08.4 uses 8 "compare" bits and 4 "filter" bits within the 32-bit address space.

The coverage and accuracy metrics are not sufficient metrics to use when making performance measurements, because they do not provide any information about the timeliness or criticality of the load misses being masked by the prefetches. What coverage and accuracy *do* provide is good feedback when tuning the prefetcher mechanism. Therefore in the framework of the simulation environment used in this paper, they are being used strictly as a means of tuning the prefetch algorithm, and should not be construed as providing any true insight into the performance of the content prefetcher.

As described in Section 3, the virtual address matching predictor has four "knobs" that can be set to control the conditions placed on a candidate address. These are compare bits, filter bits, align bits, and scan step. Figure 8 summarizes the average impact of the number of compare and filter bits on both prefetch coverage and accuracy. In this figure the prefetch coverage and accuracy values have been adjusted by subtracting the content prefetches that would have also been issued by the stride prefetcher. It is important to isolate the contribution made by the content prefetcher to properly determine a productive configuration. From the author's perspective, the decision as to which compare-filter bit combination to use is the pair of 8 compare bits and 4 filter bits, as it provides the best coverage/accuracy tradeoff.

This decrease in the coverage and increase in accuracy is explained by examining the operation of the virtual address matching algorithm. A tradeoff is being made between accuracy and coverage as the number of compare bits is increased. By increasing the number of compare bits, a more stringent requirement is being placed on candidate addresses in that more of the upper address bits are required to match. This obviously has to have a positive influence on the accuracy, as it should reduce the number of false predictions that lead to useless prefetches. The decrease in coverage comes from the reduction in the size of the prefetchable range. Assuming 32-bit addresses, using 8 compare bits leads

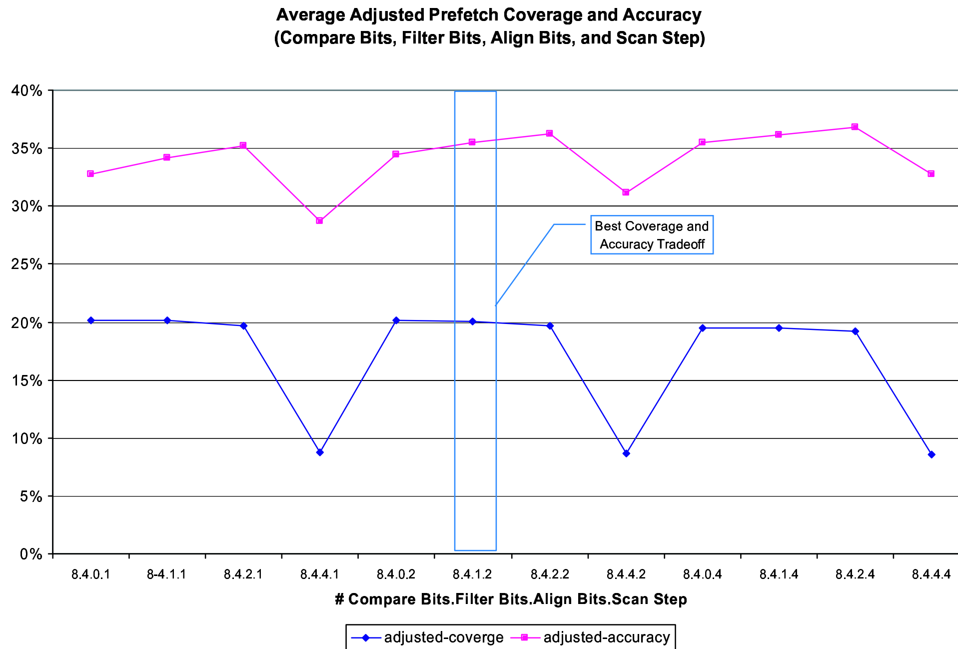

Figure 9: A summary of align bits and scan step. For example the value "8.4.1.2" means that 8 compare bits and 4 filter bits were used and that posible memory addresses in the contents were 2-byte aligned and two bytes were skipped before looking for another address.

to a prefetchable range of 16 MBytes (32 minus 8 bits equals 24 bits, which provides an address range of 0 to 16M). Now increasing the number of compare bits to 9, effectively halves the prefetchable range, which manifests itself as a decrease in the prefetch coverage.

Figure 9 summarizes the impact of varying the number of least-significant bits examined to drop candidate prefetches (alignment bits) and the number of bytes stepped after each evaluation (scan step), with the compare and filter bits fixed at 8 and 4, respectively.

Increasing the number of alignment bits to two increases the accuracy as expected, but at the expense of the coverage. This indicates that not all compilers align the base address of each node; this is expected from compilers optimizing for data footprint. For this reason, predicting only on 2-byte alignment seems the best tradeoff between coverage and accuracy. Extending this logic to cache-line scanning, the step size should be set to two bytes as pointers are expected to be at least 2-byte aligned. This is confirmed in Figure 9 where the 2-byte step size appears as the best tradeoff. This leads to a final address prediction heuristic configuration of 8 compare bits, 4 filter bits, 1 align bit, and a scan step of 2-bytes.

These bit combinations are subjective, and were chosen as such to carry out the experiments presented in the following sections. They are specific to the applications, compilers, and operating systems utilized in this study. They would require further tuning if the content prefetcher was going to be used beyond the scope of this study. One area of research currently be investigated by the authors is adaptive (runtime) heuristics for adjusting these parameters.

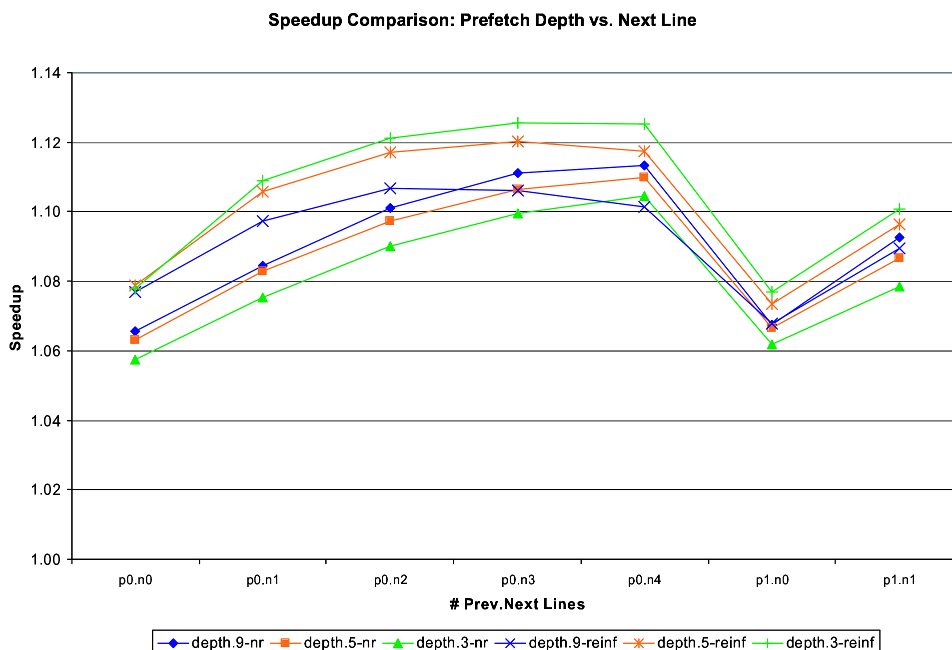

Figure 10: Speedup Summary: Prefetch Depth vs. Next Line Count. The horizontal access shows the number of previous lines and next lines that are fetched. For example, "p0.n2" means that no blocks prior to the prefetched block are requested, but two blocks following the block are requested.

4.2 Performance Evaluation

4.2.1 Speedup Comparison

Pointer-intensive applications can and do exhibit regularity, thus the logical extension to the content prefetcher is the next line prefetcher. The only concern is the added UL2 cache pollution that will occur as a result of the increase in the number of prefetches being issued. A means of compensating for this increase in requests is to decrease the prefetch chain depth threshold, that is, to trade "deeper" for "wider". A second reason for trading depth bandwidth for width bandwidth is that the deeper (or longer) the prefetch chain, the more speculative are the prefetches. Next line prefetches are not as speculative, and are more time critical. Thus the reasoning is that prefetching wider will lead to better performance.

Figure 10 provides the *speedups* for various combinations of previous and next line prefetching relative to a content initiated prefetch. Also varied is the allowed depth of the prefetch chains. When examining Figure 10, a first observation is that on the average, prefetching the previous line provides no added benefit. At the individual benchmark level, some of the benchmarks do respond positively to the previous line prefetching, but when viewed collectively, the previous line prefetching is not beneficial. This becomes more evident when making the comparison across constant bandwidth requirements. Issuing a single next line prefetch consumes the same memory bandwidth as issuing a single previous line prefetch. Examining Figure 10 shows that when next line bandwidth is exchanged for previous line bandwidth (one previous and one next, vs., two next), an overall drop in the performance is seen. In a sense this was expected. Recurrence pointers, the pointers that provide the internode connection (and the pointers that the content prefetcher tries to detect), generally point to the start address of the

node, and so the previous cache line usually has no association to the cache line pointed to by the candidate address identified by the content prefetcher.

Of interest is why when no prefetch path reinforcement is utilized, using larger prefetch depths results in better performance. As stated previously, deeper chains can lead to more speculative prefetches, and the possibility of increased cache pollution. But from a performance perspective, prematurely terminating a prefetch chain can be costly. In order to re-establish an active prefetch chain, the memory system must take a demand load miss to allow the content prefetcher to see the existing UL2 fill data, and affording it the opportunity to scan the data. So for active chains when no path reinforcement is being used, a smaller allowed prefetch depth limits the benefits the content prefetcher can provide, forcing the prefetch and execution paths to converge.

Recursive prefetch path reinforcement was discussed in Section 3. Path reinforcement provides the needed feedback required to continue uninterrupted down an active prefetch path. Seeing that the recursive paths are indeed being actively pursued, such a feedback mechansim should prove beneficial. Further examination of Figure 10 shows that path reinforcement does indeed improve performance. The important observation to be made is the reversal of which prefetch depths provide the best performance. As discussed above, when no path reinforcement is being utilzed, the larger the prefetch depth threshold, the better the performance. When reinforcement is turned on, the exact opposite is true, and the best performance is seen when the prefetch depth threshold is set at three. This occurs for several reasons. The first is path reinforcement overcomes the performance limitations that can occur when a prefetch chain is being repeatedly started, terminated, and then subsequently restarted. With reinforcement, if a prefetch chain is active it should never be terminated, regardless of the prefetch depth threshold. Next, allowing longer prefetch chains allows more *bad* prefetches to enter the memory system, as non-used prefetch chains will be allowed to live longer until being terminated. Lastly, the rescan overhead of the path reinforcement mechanism can put a strain on the memory system, specifically the UL2 cache. Allowing longer prefetch chains can result in a significant increase in the number of rescans, which can flood the bus arbiters and cache read ports, impacting the timeliness of the non-rescanned prefetches.

Figure 10 shows that the best performance is seen when path reinforcement is turned on, the prefetch depth threshold is set at three, and the content prefetches issues prefetch requests for both the predicted candidate address, and the next three cache lines following the candidate address. In this configuration, the content preftecher enhanced memory system provides a 12.6% speedup. This speedup is relative to the stride prefetcher enhanced performance simulator.

4.2.2 Contribution of TLB Prefetching

One of the benefits of the virtual address matching heuristic is its ability to issue prefetches when a virtual-to-physical address translation is not currently cached in the data TLB. The ability for the content prefetcher to issue page-walks not only leads to speculative prefetching of data values, but also leads to speculative prefetching of address translations. To measure the contribution of this TLB prefetching to the overall performance gains realized by the content prefetcher, the size of the data TLB was repeatedly doubled, starting at 64 entries, until the size of the TLB was 1024 entries. By allowing more translations to be cached, and if a significant portion of the speedups were due to prefetches to the data TLB, a marked drop in the content prefetcher speedup would be observed.

Repeatedly doubling the size of the cache from 64 entries to 1024 entries results in a small decrease in the measured speedups, only droping from 12.6% at 64 entries, to 12.3% at 1024 entries. Two

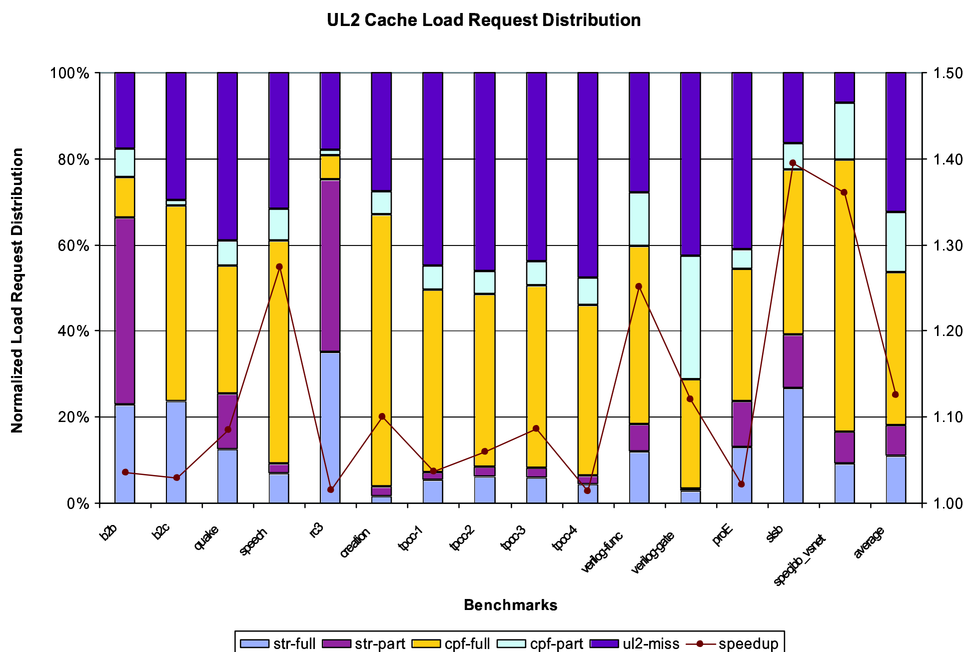

Figure 11: Distribution of UL2 Cache Load Requests.

conclusions can be drawn from the results. First, the speedups realized across the various TLB sizes remains approximately the same. This indicates that the TLB prefetching is a minor contributor to the overall performance gains being realized, and that the content prefetcher can not be simply replaced by a larger data TLB. A second conclusion is that the data TLB does not appear to be suffering from pollution caused by the speculative prefetching. If TLB pollution was a factor, an increase in speedups would be be expected as the number of TLB entries was increased.

4.2.3 Performance Summary

Figure 11 provides a look at the timeliness of both the stride prefetcher and the content prefetcher. The bottom two stacked bar provides the percentage of full and partial prefetches for stride prefetcher. The next two sub-bars are the full and partial prefetches for the content prefetcher, with the top sub-bar being the percentage of demand load misses that were not eliminated by the two prefetchers. A couple of important observations can be made. Assuming that the stride prefetcher is effective at masking the stride-based load misses, then those loads not masked by the stride prefetcher exhibit a more irregular pattern. Of these remaining non-stride based loads, the content prefetcher is fully eliminating 43% of the load misses, and is at least partially masking the load miss latency of 60% of these loads. Not all irregular loads are caused by pointer-following, and as such, the content prefetcher can not mask all the non-stride based load misses.

In the on-chip versus off-chip prefetcher placement discussion (see Section 3), one of the drawbacks expressed for placing the content prefetcher on-chip was pointer-intensive applications often have little work between pointer references. This makes it difficult to find the needed computational work to fully mask the prefetch latencies. Of the issued content prefetches that at least partially masked the memory use latency of a load request, 72% fully masked the load-use latency. This large percent-

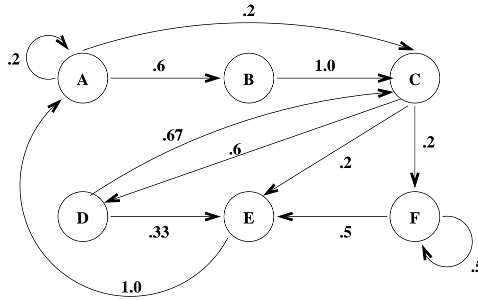

Figure 12: The Markov State Diagram for the given miss address sequence.

age is an indication that the pointer-chasing path is not a limiting factor on the workloads used in this study, and thus validates the decision to place the content prefetcher mechanism on-chip. Using the information concerning full versus partial latency masking, the content prefetcher is more timely than the stride-based prefetcher.

Overlaid on Figure 11 is the individual speedups for each of the benchmarks. For clarity, the performance results given throughout the paper have been presented using the average. Here, we show the performance of individual benchmarks.

5 Quantitative Comparison

The content prefetcher is not unique in its ability to issue prefetches in pointer-intensive applications. One prefetcher design that shares this ability is the Markov prefetcher [5]. The content-directed prefetcher may have numerous advantages over the Markov prefetcher: it uses very little state and needs no training period. In this section, we compare the performance of the content-directed prefetcher to the Markov prefetcher using the same simulation framework.

The Markov prefetcher approximates the miss reference stream using a *1-history Markov model*. Assume that a program generates the following miss reference stream, where each letter represents a different memory location that is being referenced.

A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C

The reference sequence indicates a missing reference to memory location *A*, followed by a miss for *B* and so on. Using this reference string, the corresponding Markov model is shown in Figure 12.

Each transition from node *X* to node *Y* in the diagram is assigned a weight representing the fraction of all references *X* that are followed by a reference *Y*. Assuming the program were to execute again and issue the same memory references, the Markov model could be used to predict the miss reference following each missing reference. For example, the appearance of a *D* would lead to the prediction of *C* or *E* being the next missing reference, and issue prefetch requests for *each* address.

The Markov model transition diagram is realized in hardware using a table such as shown in Figure 13. In this implementation, each state in the Markov model occupies a single line in the *State Transition Table* (STAB), and can have up to four transitions to other states. The total size of the STAB is determined by the memory available to the prefetcher. When the current miss address matches the index address in the STAB, all of the next address prediction registers are eligible to issue a prefetch.

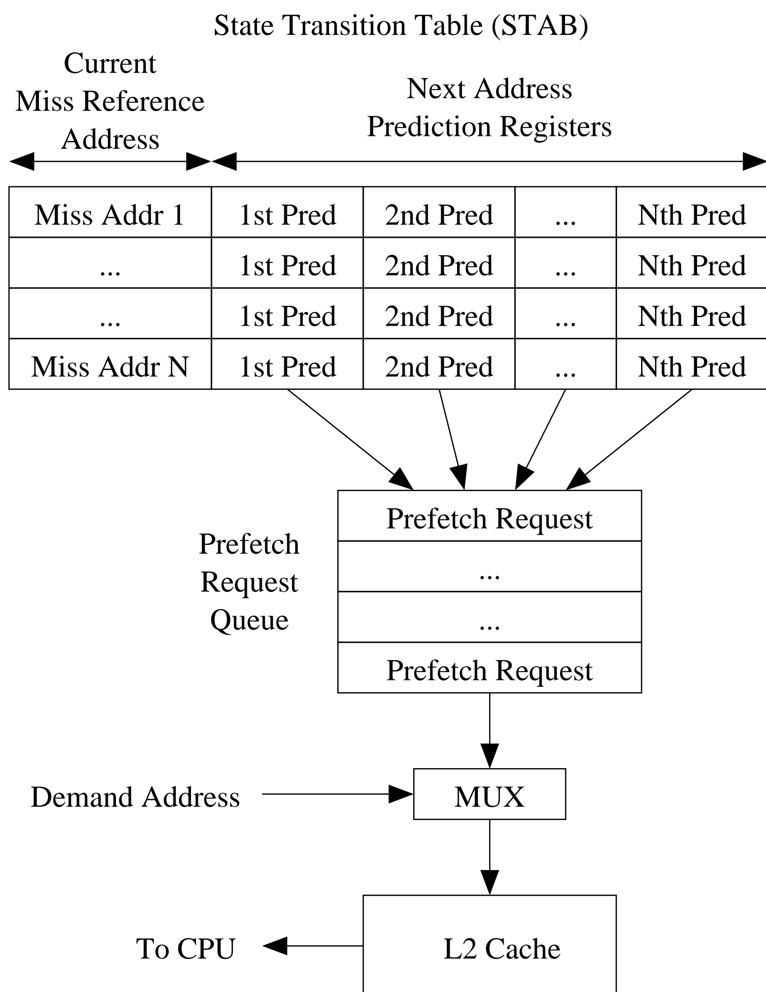

Figure 13: The Markov prefetcher State Transition Table (STAB).

Prefetch addresses are stored in the prefetch request queue and are prioritized by transition probabilities, with higher requests being allowed to dislodge lower-priority requests. The prefetch request queue contends with the processor for the L2 cache, and the demand fetches from the processor have higher priority.

The Markov prefetch mechanism used in this paper is based on the 1-history Markov model prefetcher implementation described in [5]. The prefetcher uses a transition fan out of four, and models the transition probabilities using the *least recently used* (LRU) replacement algorithm. The stride and Markov prefetchers are accessed sequentially. Precedence is given to the stride prefetcher, that is, if the stride prefetcher issues a prefetch request for the given memory reference, the Markov prefetcher will be blocked from issuing any prefetch requests. This is used to reduce the number of redundant prefetches, and provide more space in the prefetch request queue.

5.1 Performance Comparison

Two different Markov configurations were examined (see Table 3). In the first configuration, the

	1/2	1/8
Markov STAB		
size	512 Kbytes	128 Kbytes
associativity	16-way	16-way
UL2 Cache		
size	512 Kbytes	896 Kbytes
associativity	8-way	7-way

Table 3: The Markov prefetcher system configurations.

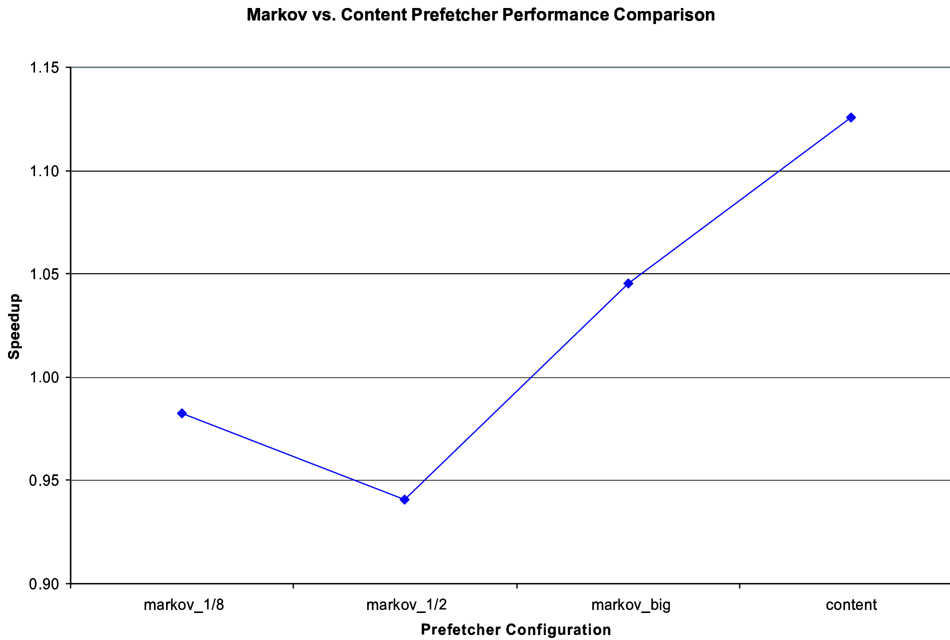

Figure 14: Comparison of the average speedup provided by the Markov and Content prefetchers.

resources allocated to the UL2 are divided equally between the Markov STAB and the UL2 cache. For the simulated processor configuration, the original 1-MByte UL2 is divided into a 512 KByte UL2 and a 512 KByte Markov STAB. In the second configuration, 1/8 of the resources allocated to the UL2 cache are re-allocated to the Markov STAB. The performance model uses an UL2 cache that is 8-way set associative, thus the 1/8 - 7/8 division of resources between the Markov STAB and UL2, respectively, is equivalent to allocating one way of each set within the UL2 cache to the Markov STAB. The resulting division of the original UL2 resources is a 896 KByte UL2 cache, and a 128 KByte Markov STAB. Resource division was used to keep the overall system resources equal between the Markov prefetcher and the content prefetcher.

The speedups provided the two memory system configurations are shown in Figure 14. These speedups are relative to the performance simulator utilizing a 1-MByte UL2 cache and enhanced with a stride prefetcher.

Also included in Figure 14 is a Markov configuration that uses a 1-MByte UL2 cache, but allows the Markov STAB to grow unbounded. This unbounded configuration, markov_big, provides an

upper limit on the performance contributions the Markov prefetcher can provide. The first two configurations, markov_1/8 and markov_1/2, clearly shows that repartitioning the original UL2 resources between the UL2 and the Markov STAB is not a wise design decision. The Markov prefetcher is not capable of compensating for the performance loss induced by reducing the size of the UL2 cache. When allowing the Markov STAB to grow unbounded (markov_big), and thus allowing resources to be added as needed to the prefetcher mechanism, the maximum possible speedup provided by the Markov prefetcher is 4.5%. The content prefetcher provides nearly three times a higher speedup, at very little cost.

The greater improvement of the content prefetcher can be explained by both the operation of the content prefetcher, and by the simulation framework. The content prefetcher does not require a training phase, which proves to be a distinct advantage. The Markov prefetcher must endure a training phase, and with large caches, such as the 1-MByte cache used in this study, there is still a strong likelihood, (modulo the application's working set size) that the data that trained the Markov prefetcher is still resident in the cache. So while the Markov prefetcher has seen the sequence previously and is now in a state to issue prefetches for the memory sequence, it is not provided the opportunity to issue prefetches as the data is still resident in the cache. Thus the training phase required by the Markov prefetcher severely restricts the prefetcher's performance potential. The content prefetcher does not suffer from these compulsory misses, and is capable of issuing prefetches while the Markov prefetcher is still training. Further, the length of the LITs may also impede the realized performance of the Markov prefetcher. Expecting the Markov prefetcher to both adequately train and provide a performance boost within the 30 million instruction constraint of the LITs may be overly optimistic.

6 Conclusions

This paper has presented *content-directed data prefetching*, a hardware-only technique for data prefetching, and simulated its performance over a number of significant applications. Also introduced is *virtual address matching*, an innovative heuristic that takes into account both memory alignment issues and bounding cases when dynamically discerning between virtual addresses and data or random values.

This study has shown that a simple, effective, and realizable content sensitive data prefetcher can be built as an on-chip component of the memory system. The design includes *prefetch chaining*, that takes advantage of the inherent recursive nature of linked data structures to allow the prefetcher to generate timely requests. The implementation was validated using a highly detailed and accurate simulator modeling a Pentium-like microarchitecture. Simulations were run on aggressively sized L2 caches, thus eliminating any false benefits that may have been realized through the use of undersized caches. The average speedup for the set of workloads was 12.6%, with the speedups of the individual workloads ranging from 1.4% to 39.5%. These speedups are very significant in the context of realistic processor designs and the fact that this improvement is in addition to that gained by using a stride prefetcher.

One of the major findings is that the content prefetcher enhanced memory system is capable of delivering timely prefetches to fully suppress demand load misses in all the applications. Previous research has indicated that this is a difficult task to achieve in the context of pointer-intensive applications.

References

[1] H-J. Boehm. Hardware and Operating System Support for Conservative Garbage Collection. In *International Workshop on Memory Management*, pages 61–67, Palo Alto, California, October 1991. IEEE Press.

[2] M.J. Charney and A.P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.

[3] T-F. Chen and J-L. Baer. Reducing Memory Latency via Non-Blocking and Prefetching Caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, Massachusetts, October 1992. ACM.

[4] J.L Hennessey and D.A. Patterson. *Computer Architecture: A Quantitative Approach. Second Edition.* Morgan Kaufman Publishers, San Francisco, California, 1996.

[5] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, Denver, Colorado, June 1997. ACM.

[6] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 388–397, 1990.

[7] M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger. SPAID: Software Prefetching in Pointer and Call Intensive Enviornments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 231–236, Ann Arbor, Michigan, November 1995. ACM.

[8] C-K. Luk and T.C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, Massachusetts, October 1996. ACM.

[9] T.C. Mowry, M.S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, Massachusetts, October 1992. ACM.

[10] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache Miss Heuristics an Preloading Techniques for General-Purpose Programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 243–248, Ann Arbor, Michigan, November 1995. ACM.

[11] S. Palacharla and R.E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, Illinois, April 1994. ACM.

[12] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, San Jose, California, October 1998. ACM.

[13] C-L Yang and A.R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 176–186, Santa Fe, New Mexico, May 2000. ACM.