Compiler-Directed Content-Aware Prefetching for Dynamic Data Structures

Hassan Al-Sukhni Ian Bratt Daniel A. Connors
Department of Electrical and Computer Engineering
University of Colorado at Boulder
{alsukhni, bratti, dconnors}@colorado.edu

Abstract

This paper describes Compiler-Directed Content-Aware Prefetching (CDCAP), an integrated compiler and hardware approach for prefetching dynamic data structures. The approach utilizes compiler-inserted prefetch instructions to convey information about a dynamic data structure to a prefetching engine. The technique eliminates the need to transform the data structure without the use of excessive prefetches and does not require prior knowledge of data traversals. The approach also eliminates the need for large hardware structures and reduces unnecessary prefetches. For pointer intensive programs, the CDCAP approach reduces memory stall time by up to 40% and out performs previously proposed prefetching techniques.

1. Introduction

Over the last decade, significant advancements in the semiconductor industry have allowed for an unprecedented increase in computer system performance. Continued exploitation of instruction level parallelism, longer pipelines and faster clocks are just a few techniques that have led to increased performance. Unfortunately, innovations in memory system design and technology have been unable to achieve the same rate of improvement. As a result, an increasing percentage of total execution time is spent waiting on the memory system. It is estimated that nearly 40% of execution time on EPIC architectures is spent stalled waiting for both instruction and data cache misses [3].

In an attempt to ameliorate the memory bottle neck, current research has explored novel mechanisms that fetch a cache line before the program issues a memory request, ideally hiding the latency of the memory system. This technique has been dubbed *prefetching*. Traditional prefetching mechanisms rely on regularity in the memory reference stream to accurately predict future memory references. Array memory accesses can be accurately predicted using hardware stride-based (linear) prefetching techniques. Applications that use dynamic data structures often exhibit data access regularity only in periodic fragments. This situation is exacerbated by the increasing popularity of object-oriented programming languages such as C++ and Java, which make significant use of Linked Data Structures (LDS). A number of hardware and software techniques have been proposed to overcome the challenge of prefetching linked data [20][4][24][25][28][10][20]. Software techniques insert prefetch instructions in the program stream [20][21][23] ahead of the actual load. Hardware techniques speculate future access addresses based on either program execution context [17][15][24], or on the contents of a cache line [11]. Only limited efforts have integrated software and run-time hardware prefetching techniques [21][25]. Recently, thread-based prefetchers [14][1] [29] have been proposed to use thread pre-computation to proceed ahead of the main thread, prefetching the main thread's data into the cache.

In this paper, we propose a novel mechanism called Compiler-Directed Content-Aware prefetching (CDCAP) to combine relevant information about program data structures with hardware prefetching systems. CDCAP builds on the strengths of both software and hardware prefetching paradigms, utilizing well established compiler techniques as well as profile information to guide the prefetching effort. The global view and understanding of linked and recursive data structures seen by the compiler combined with the dynamic run-time knowledge of a hardware prefetching system creates a promising and flexible framework for prefetching. Our technique borrows the pre-computation aspect of thread-based prefetching, but uses only limited hardware resources in the memory system to speculate ahead of program memory requests.

2. Motivation and Background

Prefetching attempts to fetch data from main memory to the cache before it is needed in order to overlap the load miss latency with useful computation. Both hardware [16][7][22][13][12] and software [18][23][8][9]

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 IEEE

prefetching methods for uniprocessor machines have been proposed. However, most of these methods focus on prefetching regular array accesses within well-structured loops or data structures with predictable access patterns. These types of access patterns are primarily found in numeric applications. Other methods geared toward integer codes [20][19] focus on compiler-inserted prefetching of pointer targets.

Linked data structures (LDS) consist of dynamically connected and allocated nodes generally taking the form of trees, graphs, and lists. In order to prefetch a specific node of the LDS, the address of that node must be obtained from an adjacent node. This sequential access of LDS nodes is referred to as the pointer traversal or pointer chasing problem. Programs tend to traverse LDS using loops or recursive procedure calls. Future references in this paper to loop traversals apply equally to recursive procedure calls. While accessing each node of the linked data structure, the program typically performs some computation on fields of the node. This computation time can be exploited by the prefetching algorithm to bring the next node into the cache system before the program requests it (by overlapping the computation time with the next node loading time). Unfortunately, the computation time within traversal loops is often much shorter than the access latency to lower levels of cache and memory system. As such, the prefetcher needs to make requests as soon as possible, even before the traversal loop is entered.

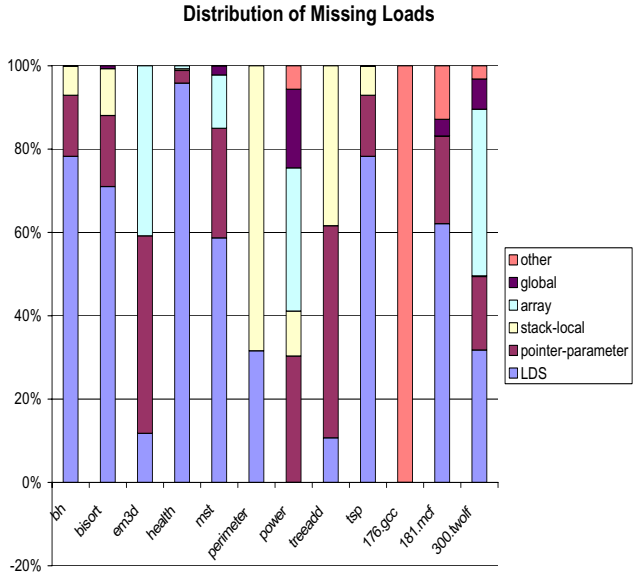

Figure 1. Dynamic distribution of missing loads.

For our analysis we chose a selection of benchmarks that make heavy use of LDS. The selected benchmarks are from the Olden benchmark suite and have been used in [24][25][6] in their evaluation of their prefetching techniques. The use of the Olden suite is augmented by a few SPEC2K benchmarks that exhibit LDS characteristics. Other benchmarks that do not exhibit large percentages of dynamically linked data structures have been removed from this paper to bring a more detailed focus on the proposed approach. Because our approach relies heavily on the compiler, programs with minimal LDS accesses will not be targeted, resulting in neither a performance increase nor decrease for those benchmarks not included in our analysis. The interested reader is refered to [5] for a complete characterization of the memory performance for the SPEC2K suite.

Figure 1 illustrates the distribution of missing loads and the degree to which linked data structure accesses deter program performance. According to the cache simulations which were run using the IMPACT compiler infrastructures, linked data structure load operations contribute to more than 30% of some program execution. It is important to note that although the benchmarks achieve nearly an average hit rate of 90%, 40% of EPIC execution is dominated by the memory system. The in-order nature of EPIC architectures hinders their ability to tolerate memory latency. Although the distribution of the LDS load operations is varied over the benchmarks, the presence of such high percentages and the increasing popularity of LDS motivates the use of a special prefetching technique for LDS loads.

2.1. Data Structure Analysis

A thorough understanding of loads that access LDS helps to facilitate a more intelligent analysis. We distinguish between the different types of loads that access an LDS: *traversal, pointer, direct and indirect*.

Traversal loads are used to sequentially traverse the LDS and represent a significan barrier to prefetching.

Pointer loads are accesses to values within the data structure that are later used as an address for a store or load. These accesses are not recursive, although other accesses can be based from the pointer data access.

Direct accesses are loads that read a value contained within the data structure. The data is used for computation operations but does not form the base address for a load or store instruction.

Indirect accesses are loads using the result of a *Pointer* load as the address. Indirect loads reference other data items from pointers other than the primary linked traversal pointer.

A basic example of the four linked load classes is illustrated in Figure 2. Figure 2(A) shows a *Traversal* load used to sequentially traverse the linked data structure. An important note is that many data structures have multiple traversal

points for reaching the next data item. Figure 2(B) describes the *Direct* access of a value contained within the data structure. Likewise, Figure 2(C) demonstrates a *Pointer* access. The data accessed lies within the data structure but is later used as an address for an *Indirect* load. An *Indirect* load is illustrated in Figure 2(D) in which the address of the load is the result of a *Pointer* load. The *Indirect* load is very difficult to prefetch in a timely fashion because it's address depends on chained pointers all having the potential of missing in the cache system.

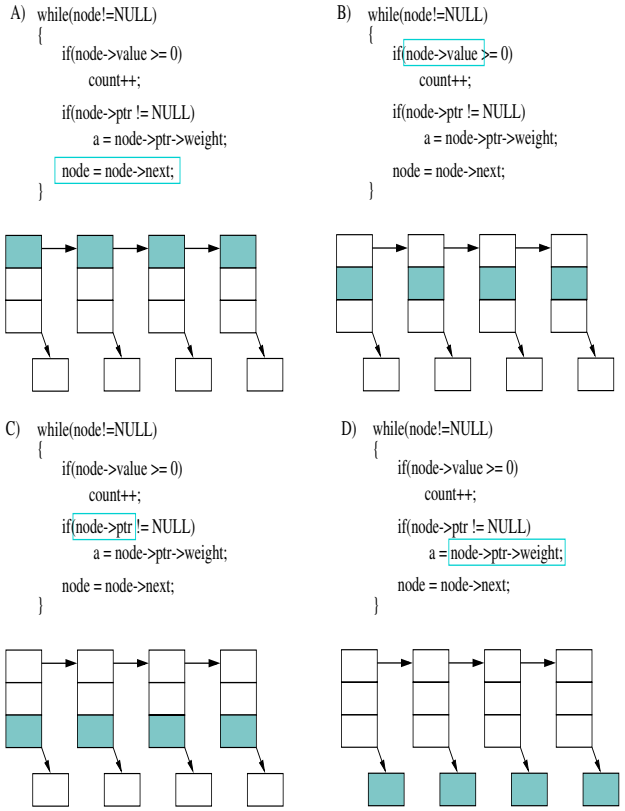

Figure 2. Basic example of four linked data structure types: *traversal, pointer, direct, and indirect*

Table 1 shows the breakdown of dynamic linked data structure load categories. These load classes are similar to the categories explored by Roth and Sohi in their evaluation of hardware jump pointers [25]. On average much of the data accesses through linked data structures consist of *Direct* and *Traversal* loads. However, benchmarks *181.mcf* and *health* use complex data structures consisting of numerous *Pointer* and *Indirect* accesses. Although several prefetching methods attempt to overlap memory accesses with other computation in the processor in an attempt to hide memory latency, few hardware systems distinguish between the different classes of loads.

Benchmark	Traversal	Direct	Pointer	Indirect
bh	32	68	0	0
bisort	47	53	0	0
em3d	19	81	0	0
health	23	0	65	12
mst	24	76	0	0
perimeter	100	0	0	0
power	0	0	0	0
treeadd	0	0	0	0
tsp	32	68	0	0
176.gcc	0	0	0	0
181.mcf	4	32	13	51
300.twolf	17	71	9	3

Table 1. Percentage of dynamic execution of linked data structure access types.

3. Related Work

We will use three metrics to evaluate the different prefetching techniques that we will address in this work; *accuracy, coverage* and *timeliness*. *Accuracy* of the prefetching technique refers to the number of usable cache lines among the total number of lines prefetched. Prefetching is a speculative effort aimed at loading values that are expected to be used by the processor. If the accuracy of the prefetcher is poor, then precious bandwidth will be waisted. *Coverage* refers to the number of program loads that are serviced by a prefetched cache line. An accurate prefetcher may cover few load misses, which results in poor enhancements to the overall system performance. The more loads that a prefetching technique can service the higher performance gains it would be capable of achieving. Software prefetching techniques are very accurate because the prefetch instructions are inserted in the proper control path and issued only where their result will be used. These techniques tend to cover good ratios of loads as well. Nevertheless, these techniques tend to acheive little overall performance gains when prefetching for LDS. This is due to the fact that little time is usually available between the issue of the prefetch instruction and the actual load. Hence, a small part of the memory latency is hidden. This aspect of prefetching is measured by *timeliness*.

In our evaluation and comparison of our work with existing techniques we use the above mentioned measures. We select two reported techniques for comparison. The first is a Content-Directed Prefetching (CDP) [11] technique which examines each address-sized content of data as it is moved

from memory to the L2 cache. Data values that are likely to be addresses are then translated and pushed to a prefetch buffer for prefetching in anticipation of their future use. As prefetch requests return data from higher memory levels, their content is also examined to retrieve subsequent candidates. This technique achieves an effect similar to thread-based prefetching by running ahead of the application. There are a number of scenarios in which the CDP will detect virtual address values that do not correspond to virtual addresses of pointers. Likewise, the hardware may prefetch virtual addresses corresponding to the linked data structure, but these may not be necessary to the code region in the vicinity of the cache miss. The second technique that we examine is Dependence Based Prefetching (DBP) [24] which works by establishing producer-consumer relationships among load instructions and using this relationship to issue a prefetch whenever the producer load is accessed. The technique is a representative of context-based prefetching techniques. DBP achieves an effect similar to software prefetching once the relationships are established. The same technique is used in a later study [25] to enhance timeliness at the expense of accuracy. Guided Region Prefetching [27], an approach similar to the one presented in this paper, was proposed at a recent conference.

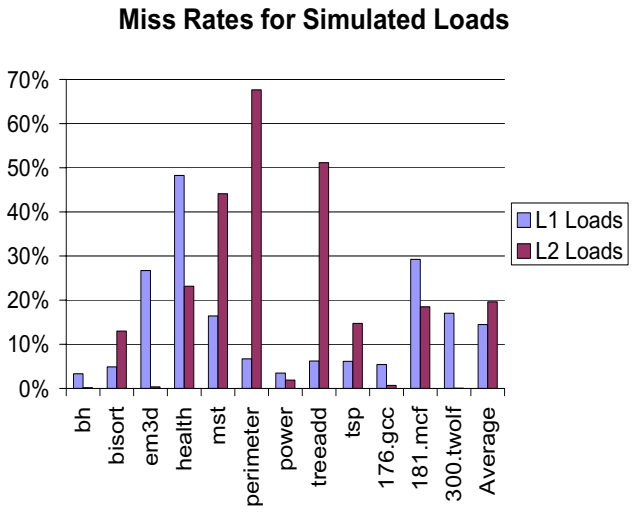

Figure 3. Loads miss rates at L1 and L2 caches

Figure 3 shows L1 and L2 miss rates for a selected number of benchmarks. To the best of our knowledge, all existing prefetching techniques are designed to work at a specific level in the memory hierarchy. It is clear from the figures that the benchmark *em3d* is best suited for prefetching into the L1 cache. On the other hand, *perimeter* and *treeadd* are expected to benefit from L2 prefetching techniques. *Health* and *mst* may benefit from both. The rest of the benchmarks have moderate chances for improvement from both L1 and L2 prefetching.

DBP is designed to issue prefetches at the L1 level only. This is because DBP issues prefetches within one iteration of the traversal loop. The technique cannot iterate speculatively ahead of the main program because it uses the main program loads to trigger further prefetching. The ability of DBP to produce timely prefetches depends on the computation time that is available within the traversal loop body, or between the producer and consumer loads in general. Because the amount of computation to be done within a loop is often small, the DBP prefetching scheme naturally caters toward prefetches with relatively low latency (L1). On the other hand, CDP is designed for a high degree of speculation resulting in a greater strain on the memory bandwidth and forcing the use of the CDP technique at the second level or even the third level in the memory hierarchy. The amount of traffic caused by a CDP at the first level of cache will negatively impact performance. This was confirmed by our study and evaluation.

4. CDCAP

Our Compiler-Directed Content-Aware Prefetching (CDCAP) technique evolves the idea of Content-Directed Prefetching (CDP) [11] by combining a set of hardware components collectively called the Hardware Prefetching Engine (HPE) with compiler inserted instructions that eliminate the need to detect dynamic data structures in prefetched cache lines. Also related to our approach is a recent prefetching scheme [26] that designates a more sophisticated memory system controller outside the scope of the processor for analyzing memory access requests and making prefetches. In contrast to the hardware CDP prefetching scheme and the memory controller prefetcher, our CDCAP approach statically identifies LDS locations and uses profile information to construct prefetch instructions that direct the HPE. The prefetch instruction is interpreted into a set of tasks that are then sent to a controller called the *Linked Prefetching Control Engine*. Overall, the proposed prefetching architecture mechanism is depicted in figure 4 and consists of the following components:

Content-Aware Prefetching Buffer(CAPB) The CAPB is a dedicated cache that stores prefetched data. This cache is accessed simultaneously with the main cache that the HPE is attached to. If requested data is available in the CAPB and misses in the main cache. The request is serviced from the CAPB and the main cache line is filled accordingly. The CAPB shares the same bus into the next memory level with the main cache. The HPE is allowed to issue prefetches only during idle bus cycles.

Linked Prefetching Control Engine (LPCE) The prefetching hardware component controlled by the compiler directives and the CAPB contents to direct future prefetches based on returned cache contents.

Static Data Structure Prefetching Instruction, *the Harbinger* The *harbinger* is used for conveying static data structure layout information to the LPCE. Each instruction has a unique ID. Harbinger instructions are stored in a dedicated table in the HPE, shown in the figure as HIS, Harbinger Instructions Store.

Dynamic Invocation Hints EPIC style architectures allow the compiler to insert hints within instructions to convey information to the memory system. Hints are inserted into load instructions that bring the LDS base address from memory. These hints are consumed by the LPCE. The LPCE upon receiving the ID of a previously received prefetch instruction, will start a recursive prefetch effort. The effort is based upon the information contained in the prefetch instruction and the loaded address of the hint-specifying load.

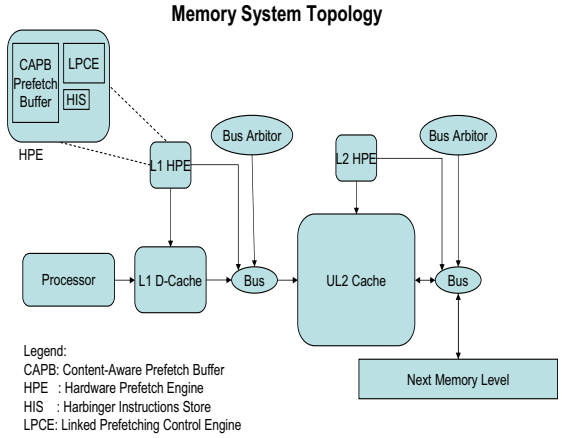

Figure 4. Content-Aware Prefetching Memory System.

Similar to hardware-based prefetching, prior to issuing a prefetch request, the main cache and system bus arbiters are checked for any potential outstanding block requests. The HPE keeps a counter for each prefetch instruction initiated in order to prefetch the estimated depth of the LDS. The engine uses the compiler inserted directive to maintain a request mapping for the incoming contents brought from the memory system. This encoding invokes further prefetches based on the retrieved contents. Hence, achieving an effect similar to what thread-based prefetching techniques try to. The advantage of CDCAP over thread-based techniques lies in the ability of CDCAP to initiate traversal prefetches directly after the node is placed at a specific level in memory. Thread based prefetching cannot issue a traverse until the node is placed in the highest level of memory and processed by the speculative thread.

4.1. Content Prefetching Directives

The CDCAP prefetching approach involves the introduction of a new instruction, the *prefetch harbinger*:

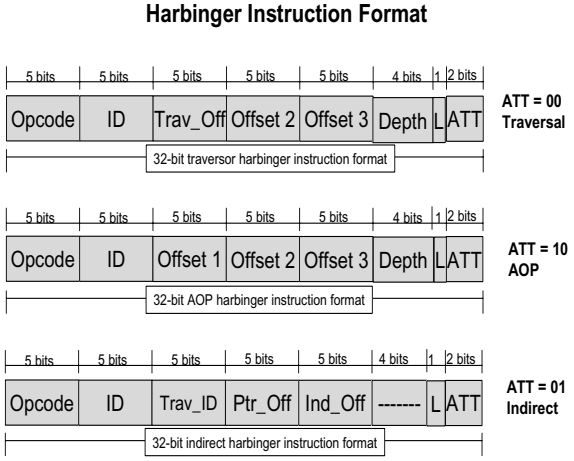

Figure 5. Compiler-directed *prefetch harbinger* instruction.

Figure 5 illustrates the harbinger instruction used to communicate the exact nature of the linked data structure to the the HPE. The new extensions designate different aspects of how the data structure is used in an upcoming LDS traversal loop and the degree to which the prefetching engine should initiate recursive prefetches. Alternatives to using new extensions are part of on-going research and are not explored in this initial study.

The proposed prefetching instruction consists of five fields:

Harbinger ID a unique ID that is associated with each harbinger instruction.

Access Type Template The access type template indicates the type of the harbinger instruction. We recognize three types.

Traversal Harbinger describes the offset of the linking address within the data structure along with additional pointer loads that will be accessed within the traversal loop. These offsets are included in the *Offset Vector* field. This harbinger serves for traversing the nodes of the data structure along with associated pointer loads.

Array Of Pointers(AOP) Harbinger is designed to assist prefetching arrays of pointers. The first Offset field in the offset vector is similar to the increasing offset of arrays. They may contain pointers and any other data types. The location of the pointer within the array is encoded in the second offset of the harbinger.

Indirect Harbinger informs the HPE of an indirect load that needs to be prefetched with each node. The load referred to as *Access 3* in Figure 5 is of this type. The indirect harbinger has a field for the associated traversal harbinger in *Trav_ID*, and the relevant pointer offset within the traversal harbinger in the *Ptr_Off* field. The LPCE will control the invocation of related harbinger instructions at every iteration of the recursive traversal.

Offset Vector The contents of data retrieved from memory due to a request initiated from either the compiler-inserted hint or subsequent recursive prefetches are analyzed with respect to the offset vector. The vector in conjunction with the access type template is used to determine if the linked data structure requires greater spatial prefetches (next line or previous line) or if a pointer/traversal type is necessary.

Recursion Depth The depth field is used to inform the HPE as to the expected number of traversals across different iterations of loops. The depth is a weighted encoding that allows a large range of prefetch iterations to be made. This depth is determined based on profile runs. The depth field is used by the HPE to guide the recursion. CDP uses a path reinforcement measure to control its recursion. Cooksey reports in [11] that the path reinforcement had minor effect on his prefetcher performance.

Level (L) Indicates the level in the cache hierarchy to apply the recursive prefetch.

The instruction overcomes the problems of base CDP that indiscriminately speculates that every potential data item that appears as a virtual address will be relevant to future execution.

4.2 Linked Data Structure Example

Figure 6 contains code generated from the function *check_patients_waiting* in the benchmark *health*. Within *check_patients_waiting*, a while loop iterates through a linked list until reaching the end of the list. Each node in the list contains a pointer to another data structure as well as a pointer to the next node. Simulation results show that miss penalties caused by the four highlighted loads account for approximately 80% of the total time spent stalled on data memory operations. Upon entry to the section of code, r2 contains the head pointer of the list. Consequently, our prefetching algorithm will schedule the harbinger instruction directly following CB2. The while loop begins by loading a pointer from the linked list (op 10) to be later used as an address for op 12. Op 10 is therefore classified as an *Pointer* load while op 12 is classified as *Indirect*. Op 15 traverses the linked list and is classified as the *Traversal* instruction.

Figure 7 illustrates three iterations of the address request pattern for the previous example of the *health* benchmark. Each iteration has three accesses, a *traversal*, *pointer*, and an *indirect*. The CDCAP issues prefetches starting from the first *traversal* access during the first iteration, subsequently a *pointer* will be recursed in the second iteration, followed by an access to an *indirect* data item. These prefetches are

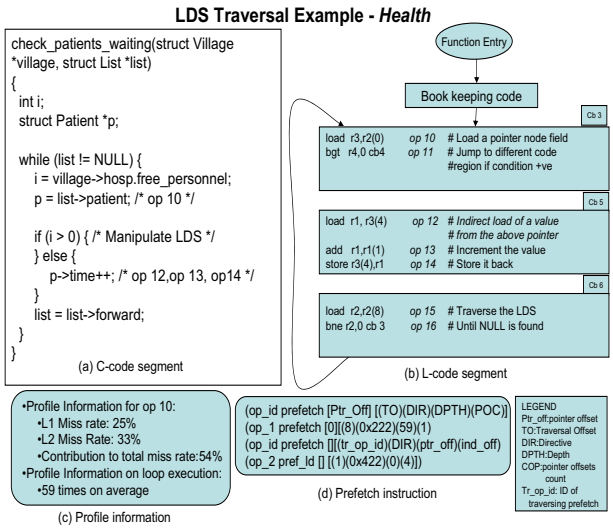

Figure 6. Source and low-level code example of linked data structure in Olden's *health*.

invoked via the prefetch instructions. It is important to compare this example to the other techniques. In CDP, the access to the contents of the first cache line will initiate accesses (2) and (4). In addition, depending on the other cache line, it may also issue prefetches to unnecessary addresses found in the cache line. A similar event happens at the access of (2), in which more unnecessary prefetches may be issued.

In the CDCAP approach, these erroneous prefetches are eliminated by informing the layout of the data structure to the hardware using the *harbinger* instruction. In comparison to the dependence-based approach (DBP), there are several disadvantages. First, DBP must have previously traversed the data structure nodes and have also built a full dependence table between all of the pointer connections in the above example. The dependence table for this example would contain information correlating accesses (1 to 2), (1 to 4), (2 to 3), etc, requiring all iterations of the loop to be represented in the dependence structure. On the other hand, CDCAP is comparatively a stateless approach to maintaining the connection between data structure accesses since it does not maintain information about past traversals and pointers.

5. Experimental Evaluation

5.1. Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support a model of an EPIC architec-

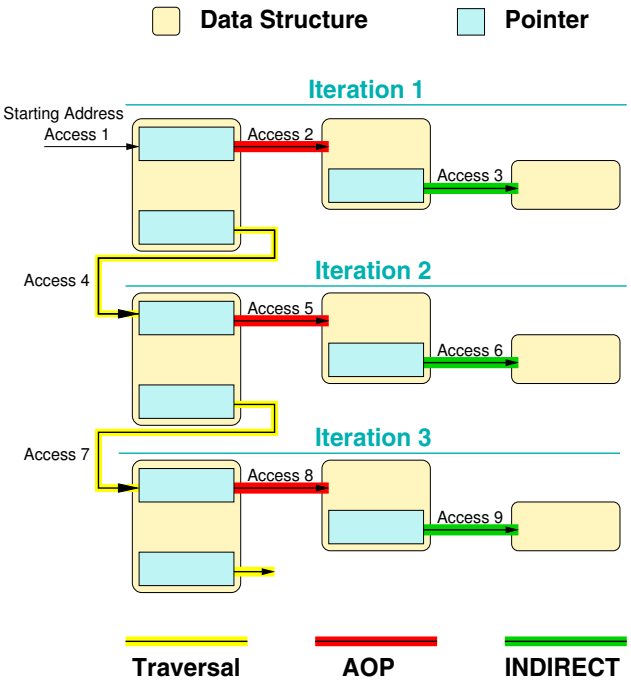

Figure 7. Memory accesses and prefetching commands for CDCAP on Olden's *health*.

ture [2]. The base level of code consists of the best code generated by the IMPACT compiler, employing function inlining, superblock formation, and loop unrolling.

The base processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and three branch units. The instruction latencies used match the Itanium microprocessor (integer operations have 1-cycle latency, and load operations have 2-cycle latency.) The execution time for each benchmark was obtained using detailed cycle-level simulation. For branch prediction, a 4K entry BTB with two-level correlation prediction with a branch misprediction penalty of eight cycles is modeled. The parameters for the processor include separate 32K direct-mapped instruction and data caches with 32-byte cache lines with a miss penalty of 10 cycles. A 1M 8-way set associative with 128-byte cache lines second-level unified cache (with memory latency of 130 cycles) is also simulated. Each cache has its own hardware prefetching engine. The base prefetch engine runs stride prefetching. The stride prefetcher uses 16-entry pc-indexed table with a threshold parameter to start stride prefetching. All prefetch implementations use a dedicated prefetch buffer.

DBP was simulated with an 8KB L1 Prefetch buffer with 32 byte cache lines and 4-way set associativity. Content-Directed data Prefetching was implemented using an L2 Prefetch Buffer of 256KBytes with 128 bytes cache lines and 4-way set associativity. Path reinforcement mechanism of the CDP technique was not simulated, this reinforcement contributes less than 1.3% of the performance.

The CDCAP approach was implemented with the above configuration for L1 and L2 prefetch buffers attached to both L1 data cache and the unified L2 data cache. The technique was applied to L1 for *bh* and *em3d* and to both L1 and L2 in *health*. The rest of the benchmarks had CDCAP applied to L2 only. All evaluated techniques were queueing prefetch requests to the same prefetch queue in each level of the cache. The bus arbitration policies were the same for the different simulations. Benchmark simulations ran up to a maximum of 100 million cycles, or completion whichever happened first. Initial warm up statistics were not discounted.

5.2. Results and Analysis

The proposed Compiler-Directed Content-Aware Prefetching technique promises to achieve a working balance between the three measures that are used for evaluating the prefetching techniques, *accuracy, timeliness and coverage*. The ability of the architecture to prefetch into the different levels in the memory hierarchy is a key advantage. To evaluate the techniques under study, different memory system measures are reported.

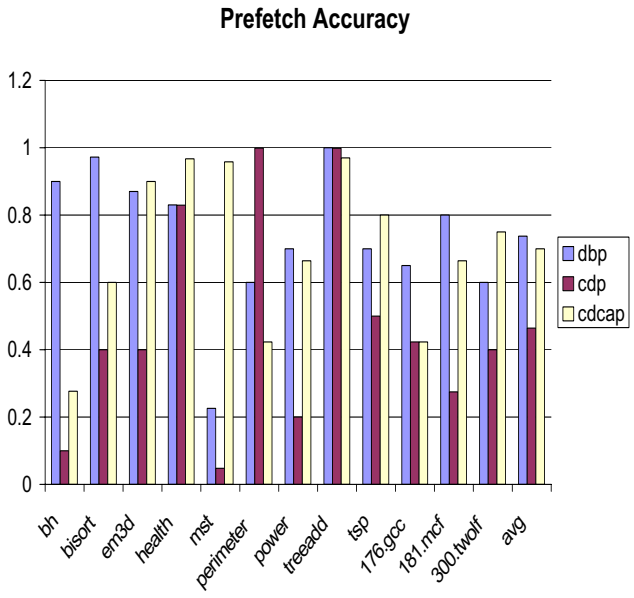

Figure 8. Prefetching accuracy.

The prefetching accuracy is defined as the ratio of used blocks to the total number of cache blocks prefetched. Figure 8 shows the prefetching accuracy for the three studied

techniques. The graph supports our earlier discussion about the added accuracy of software oriented prefetching techniques. Such techniques can use program control flow and execution path to reduce unnecessary prefetches, and hence have higher accuracy on average. This knowledge is not as accessible to hardware prefetching techniques like CDP. The more aggressive the prefetching technique the less accurate it becomes. Our proposed CDCAP seems to score close to DBP in accuracy. Prefetching techniques used at higher levels in the memory hierarchy need to be more accurate. Otherwise they will be wasting precious bandwidth.

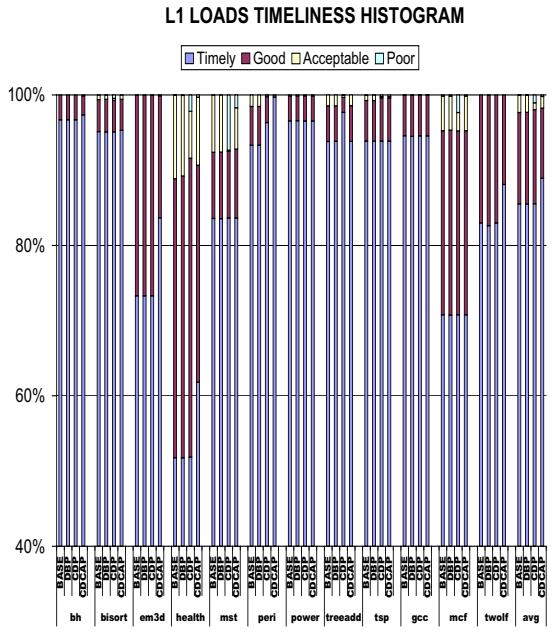

Figure 9. Timeliness of prefetching.

Figure 9 shows our measure of the timeliness of each of the prefetching techniques. Each load that arrives at the L1 data cache will eventually be serviced and sent back to the processor. The time in cycles that it takes the memory to service each load was measured. We show in each bar of the graph four slices, the bottom slice refers to loads that were serviced within one quarter of the memory latency and are labeled *Timely*. Loads that were serviced within half the memory latency are labeled *Good*. Loads serviced within the third quarter are labeled *Acceptable*, and finally those that were serviced in times more than that are collectively labeled *Poor* and occupy the upper part of each bar. This histogram indicates that the timeliness of our CDCAP technique is superior to the existing techniques. It also says that DBP was not timely enough, because it did not enhance the response time of the memory system. CDP seem to have different effects on different benchmarks. For *health, perimeter* and *treeadd* CDP does well. These benchmarks tend to have multiple addresses in one node, and tend to access them within short periods of time. This makes aggressive prefetching pay off. However, for the rest of the benchmarks CDP hurt performance by extending the last quarter of the bar.

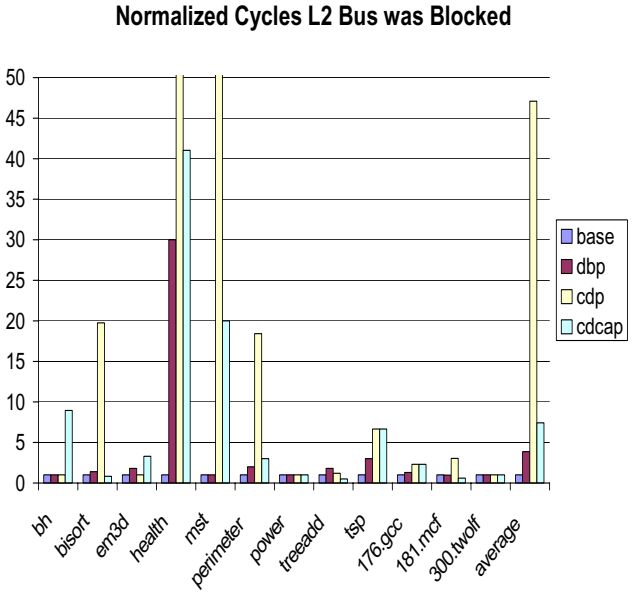

Figure 10. Normalized bus blocking with prefetching.

Figure 10 shows the normalized time in simulated cycles that the lower memory level bus was busy when needed for the different evaluated prefetching techniques. The high number that CDP incurs proves our argument. This is due to the fact that the algorithm attempts to issue prefetches too aggressively even with the several heuristics that Cooksey uses to limit the number of issued prefetches. On the other hand, DBP seems to be the least intrusive in issuing prefetches which affect other loads. Although CDCAP seems to be very intrusive in some benchmarks like health, this is not all bad, because the accuracy of the technique implies that this occupation of the bus is not going to be wasted. Figure 10 suggests that the three prefetching techniques are not issuing prefetches when the bus is idle. This implies that an effective prefetching technique must limit the amount of unnecessary prefetches. On average, CDCAP seems to achieve a working balance.

Performance measured as the normalized cycles of processor stalls waiting for memory is reported. Figure 11 shows clearly that balancing the prefetching technique has

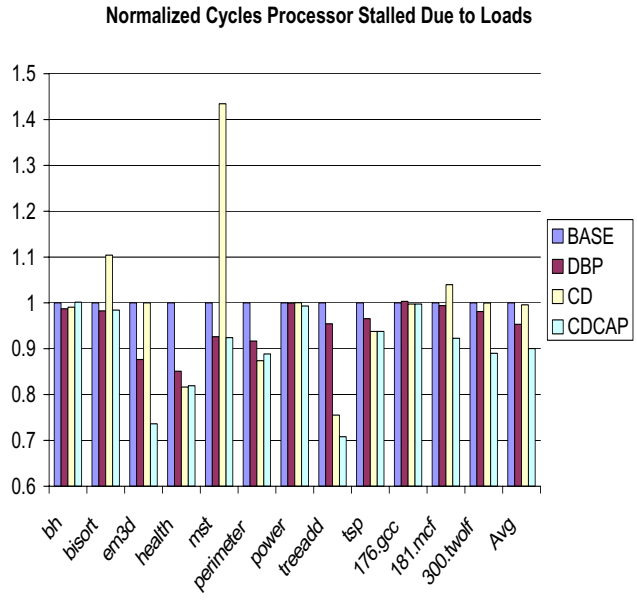

Figure 11. Normalized cycles of processor waiting for load values.

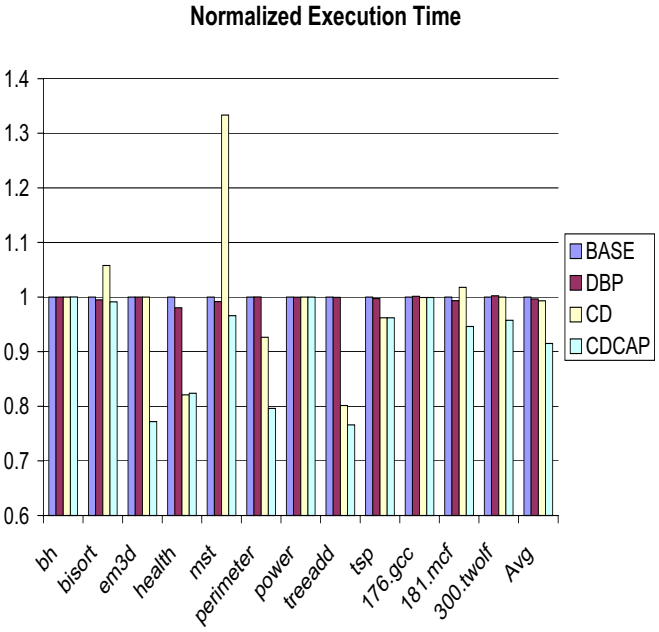

Figure 12. Normalized execution times.

an important role to play in prefetching. Ranges of performance improvement relative to the base architecture model with stride prefetching indicate that on average CDCAP reduces stall time by 10%. The largest improvements are for *health* and *treeadd*. Improvements are made in most benchmarks, and CDCAP consistenly out-performs the hardware Content-Directed Prefetching.

Lastly, normalized execution times of the simulated benchmarks is shown in figure 12. Execution speedup is directly related to the number of cycles a processor needs to stall waiting for data from load instructions. The figure shows speedups of up to 24% and an average of 9%. This again stresses the fact that CDCAP consistently outperforms evaluated prefetching techniques.

6 Summary

As cache penalties continue to have a significant impact on memory system performance traditional hardware-based prefetch techniques will not evolve sufficiently to help with accesses to linked data structures. By integrating compiler and hardware techniques in a coordinated approach, the cache penalty effects for accessing recursive data structures can be reduced. With Compiler-Directed Content-Aware Prefetching approach presented in this paper, a memory system can be improved by an average 10%, while reducing the bandwidth requirements of content-aware prefetching by a factor of 6. Although these results have only been highlighted for applications with large amounts of linked data structures, the potential of the approach is very wide reaching. Likewise, the system illustrates that certain prefetching methods are not suited for higher levels of cache due to bandwidth limitations.

The area of future work is to continue to develop other intelligent prefetch mechanisms that are guided by compiler designation of the connection of linked data structures. Similarly, the approach will be evaluated for C++ and Java programs to understand how content-aware prefetching can benefit performance. Lastly, by applying the technique to instruction memory requests, we hope to explore how the content-aware technique can aid prefetching of upcoming control paths. A likely next step will be to further evaluate the hardware complexity of the content-aware prefetcher and conduct further studies to tune and validate designs. Finally, these mechanisms will need to be examined in the context of a multiprocessor and multithreading systems.

References

[1] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in

the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] I. Bratt, A. Settle, and D. A. Connors. Predicate-based transformations to eliminate control and data-irrelevant cache misses. In *Proceedings of the First Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques*, pages 11–22, December 2001.

[4] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 2001 international conference on Parallel Architectures and Compiler Technology*, pages 52–63, 2001.

[5] J. F. Cantin. Cache performance for spec cpu2000 benchmarks. *Computer Architecture News*, 29(4), September 2001.

[6] A. R. M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. In *ACM Transaactions on Programming Languages and Systems*, March 1995.

[7] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[8] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 69–73, November 1991.

[9] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang. Tolerating data access latency with register preloading. In *Proceedings of the 6th International Conference on Supercomputing*, July 1992.

[10] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 1–12, 1999.

[11] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 201–213, October 2002.

[12] F. Dahlgren, M. Dubois, and P. Strenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 56–63, August 1993.

[13] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proc. 25th Ann. Conference on Microprogramming and Microarchitectures*, Portland, Oregon, December 1992.

[14] B. C. J. Collins, S. Sair and D. Tullsen. Pointer cache assisted prefetching. In *International Symposium on Microarchitecture*, November 2002.

[15] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[16] 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 International Symposium on Computer Architecture*, pages 364–373, May 1990.

[17] P. C. K. Farkas and Z. Vranesic. Memory system design considerations for dynamically scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

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

[19] M. H. Lipasti, W. J. Schmidth, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 231–236, December 1995.

[20] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, 1996.

[21] F. D. M. Karlsson and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *The 6th International Symposium on High-Performance Computer Architecture*, January 2000.

[22] S. Mehrotra and L. Harrison. Quantifying the performance potential of a data prefetch mechanism for pointer-intensive and numeric programs. Technical Report 1458, Center for Supercomputing Research and Development, University of Illinois, November 1995.

[23] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct 1992.

[24] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126, 1998.

[25] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 111–121, 1999.

[26] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, May 2002.

[27] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[28] C.-L. Yang and A. R. Lebeck. Push vs. pull: data movement for linked data structures. In *Proceedings of the 14th international conference on Supercomputing*, pages 176–186, 2000.

[29] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, June 2001.