AC/DC: An Adaptive Data Cache Prefetcher

Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith
University of Wisconsin –Madison
Department of Electrical and Computer Engineering
{kjnesbit, dhodapka, jes}@ece.wisc.edu

Abstract

AC/DC is an adaptive method for prefetching data from main memory. The basic prefetch method divides the memory address space into equal-sized concentration zones (CZones), and uses a global history buffer to track and detect patterns in miss address "deltas" (differences between consecutive addresses) within each CZone. When simulated with a realistic desktop memory system, CZone prefetching with Delta Correlations (C/DC) outperforms four other previously proposed prefetching methods. C/DC yields an average performance improvement of 23 percent when compared with no prefetching.

Adaptivity is then added to the basic method. A tuning algorithm dynamically configures the CZone size and prefetch degree (i.e. the amount of data prefetched) on a per program-phase basis. Adaptive reconfiguration provides additional performance improvements of 4% over C/DC. Overall, the Adaptive CZone / Delta Correlation (AC/DC) method outperforms other methods studied by 10%.

1. Introduction

Over the past two decades, advances in semiconductor process technology and microarchitecture have led to significant reductions in processor cycle times. Meanwhile, advances in memory technology have led to ever increasing memory densities, but relatively minor reductions in memory access times. Consequently, memory latencies measured in processor clock cycles are continually increasing and are now on the order of hundreds of clock cycles. Cache memories help bridge the processor-memory latency gap, but caches are not always effective.

One of the basic techniques for enhancing cache performance is prefetching. As the processor-memory latency gap continues to increase, there is a continuing need for development and refinement of prefetch methods. In this paper, we propose an innovative method for cache prefetching aimed specifically at prefetching from main memory.

The proposed prefetching method uses concentration zones (CZones) [19] that divide memory into fixed size zones. Then, the prefetcher looks for stride patterns in sequences of cache misses directed toward the individual zones. When it finds an access pattern within a CZone, it launches prefetch requests. This method has the desirable property of not needing program counter values for the load instructions that cause misses. Lower levels of the memory hierarchy are often far removed from the processor core (and may even be off-chip) so the program counter is typically not readily available [12][16][19].

CZone history information is held in a Global History Buffer (GHB) [18]: a FIFO structure that stores all recent L2 cache miss addresses in the order in which they occur[1]. In the GHB, miss addresses within the same CZone are placed in a time-ordered linked list. This is in contrast to conventional cache miss history tables, which are directly addressed. The GHB method has the advantage of yielding a small prefetch structure that automatically flushes itself of stale prefetch information.

The proposed prefetcher uses delta correlations [15][18] to prefetch miss address patterns that are more complex than simple strides. This method is referred to as CZone / Delta Correlation Prefetching (C/DC). Looking beyond simple strides is particularly important for CZone prefetching because CZones are not completely effective at isolating access locality (discussed further in Subsection 3.2).

In general, performance of CZone prefetchers are sensitive to the CZone size, and optimal CZone size is a program dependent characteristic. In particular, the CZone size should roughly match the size of the data structures that are being accessed. Prefetch performance is also sensitive to prefetch aggressiveness or prefetch degree (the number of prefetches triggered by

[1] For simplicity, in this paper we assume the lowest level cache is the L2.

a single prefetch event). A high prefetch degree can significantly improve the performance of some programs, but the same prefetch degree can cause memory contention that degrades the performance of other programs. Consequently we use an adaptive tuning algorithm that operates along two dimensions – for each program phase it dynamically adjusts both the CZone size and the prefetch degree to find a near-optimal configuration. This enhanced method is referred to as Adaptive CZone / Delta Correlation Prefetching (AC/DC).

We evaluate C/DC and AC/DC prefetching via simulation of a modern high-end desktop system running SPEC CPU2000 benchmarks. C/DC and AC/DC are shown to outperform a number of previously proposed prefetching methods. C/DC outperforms previously proposed methods by as much as 15%, and performs 6% better on average. For a subset of the SPEC benchmarks (chosen for its diversity, and including benchmarks where prefetching tends to hurt performance), AC/DC is shown to outperform C/DC by an additional 4%.

2. Related Work

2.1. Stride Prefetching

Stride prefetching techniques detect sequences of addresses that differ by a constant value, and launch prefetch requests that continue the stride pattern [6][9][13][16][19][25]. The simplest methods prefetch only unit strides, i.e. where addresses are one location apart. Some early methods indiscriminately prefetch sequential lines [13], while other methods wait for a sequential access stream to be detected before prefetching [25].

More advanced stride prefetching methods can prefetch non-unit strides by storing stride related information in a history table [6][16][19]. Each table entry 1) holds the most recent stride (the difference between the two most recent preceding addresses) 2) the most recent address (to allow computation of the next stride), and 3) other state information that determines conditions under which a prefetch should be triggered. When the current address is a and a prefetch is triggered, addresses $a+s$, $a+2s$, . . ., $a+ds$ are prefetched – where s is the detected stride and d is the prefetch degree; more aggressive prefetch implementations will use a higher value for d. Arbitrary Stride Prefetching [6] was one of the first schemes for stride prefetching, but uses the program counter as a table index, making it less feasible for prefetching into secondary caches.

Stride Stream Buffer CZone Prefetching [19] is a stride prefetching method that does not use program counter values for table indexing. Instead, CZone prefetching partitions the memory address space into fixed-size (power-of-two) CZones. Two memory references are in the same CZone if their high-order n-bits, the CZone tag, are the same. The value of n is an implementation-specific parameter. CZone prefetching uses the CZone tag to access a filter table for detecting constant strides among addresses within each CZone.

2.2. Correlation Prefetching

Correlation Prefetching methods look for address sequence patterns in order to predict future cache behavior. Markov Prefetching [12] correlates global miss addresses. Distance Prefetching [15] was originally proposed to prefetch TLB entries, but was adapted in [18] to prefetch cache lines. The adaptation correlates deltas (differences in consecutive addresses) in the global miss address stream. Tag Correlation Prefetching [10] is a two-level correlation prefetching method that also uses the global miss address stream. The conventional cache index accesses a first level tag history table (THT). The THT contains the last n tags with the same cache index. These tags are combined to access a second level Pattern History Table (PHT). The PHT holds the next predicted tag, which is combined with the cache index to generate a prefetch.

Correlation prefetching has two advantages. First, correlation methods can prefetch access patterns beyond constant stride access patterns. Second, correlation prefetching methods often do not require the program counter to detect access patterns.

2.3. Prefetching with a Global History Buffer

Most history-based prefetch methods use a table that is directly addressed using an index value. An alternative structure uses a *Global History Buffer* (GHB) [18] (See Figure 1). The GHB is an n-entry FIFO table (implemented as a circular buffer) that holds the n most recent L2 miss addresses. Each GHB entry stores a global miss address and a link pointer. The link pointers chain the GHB entries into time-ordered address lists. An *Index Table* (IT) holds the initial pointers to the linked lists. The index table is accessed via some key; depending on the key that is used, a number of history-based prefetch methods can be implemented. In a recent study by Pérez et al.[20], ten prefetch mechanisms proposed since 1990 were compared using a standard simulation framework; the GHB method gave the best performance.

In earlier work [18] the GHB was used for implementing both stride and correlation prefetching methods. However, in that work, CZone indexing of the GHB was not considered. In this paper we use the CZone tag to access the Index Table.

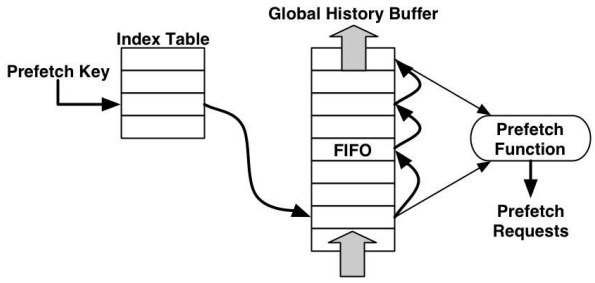

Figure 1: Global History Buffer Prefetching

3. CZone Delta Correlation Prefetching

We begin by defining and evaluating the CZone / Delta Correlation (C/DC) prefetcher, and then add adaptivity (AC/DC) in Section 5. As described above, CZone prefetching divides memory into fixed size zones, typically powers of two. Then, it looks for patterns within each zone. Using delta correlation enables prefetching of more complex patterns than simple strides. For example, consider correlation on pairs of consecutive deltas in the address sequence shown below (time goes to the right).

Addresses	47	49	54	56	58	63	65
Deltas		*2*	*5*	*2*	*2*	*5*	*2*

In this example, the two most recent deltas are 5 and 2. If the address sequence is searched in reverse order for the same delta pattern, it is first found at (49, 54, 56). When the delta pair (5, 2) appeared previously, the next address deltas were 2 and 5, therefore, if the prefetch degree is two, a prefetch for addresses 67 and 72 is triggered (65+2, 67+5).

In general, the sequences of deltas used for correlation can be any length. We undertook a preliminary study that indicated that pairs of deltas are a good choice. Using three deltas provided an insignificant improvement in prefetch accuracy (i.e. the percent of prefetches that are actually used by the program before being evicted) over delta pairs, and in some cases degraded prefetch coverage (i.e. the percent of memory accesses that are prefetched rather than demand fetched) and performance. Consequently, in the remainder of this work, we consider correlations using pairs of address deltas as in the example given above.

3.1. Implementation Details

The global history buffer (GHB) used in the C/DC prefetcher organizes the most recent L2 miss addresses into linked lists, with the Index Table holding the head pointers of the lists. To this basic structure, we add a small delta buffer, a correlation key register, and a correlation comparison register (See Figure 2).

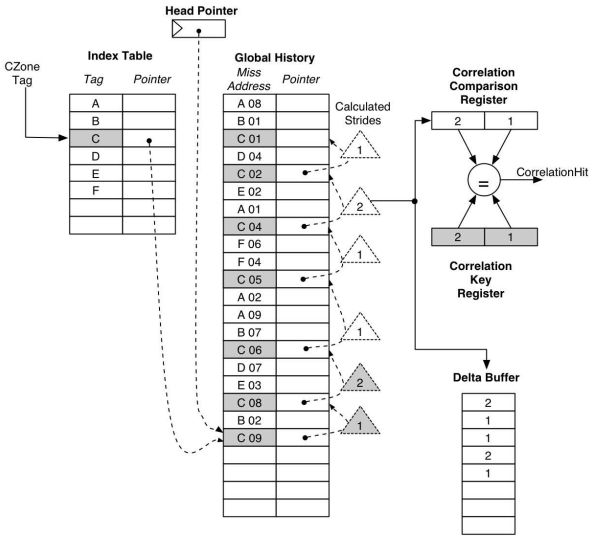

Figure 2: C/DC prefetching hardware with an example recurring access pattern.

A simple state machine controls the C/DC prefetching algorithm's GHB accesses. The state machine starts when a load misses the L2 cache. The CZone tag (the high order n-bits) of the miss address is used to access the Index Table. If the CZone tag hits in the Index Table, the Index Table entry points to the list of preceding miss addresses that are in the same CZone.

The correlation mechanism then compares pairs of deltas in the miss address stream. The first delta in the address sequence, i.e. the difference between the current miss address and the miss address at the head of the linked list, is computed and shifted into the correlation key register. On the next cycle, the next element of the linked list is accessed, and the second delta is computed and shifted into the correlation key register. This second delta is also shifted into the correlation comparison register. After the second delta, the linked list is walked and deltas are shifted into the correlation comparison register (with the older value being shifted out). At each step the comparison register and the key register are compared. If the registers match, a correlation has been detected and prefetching is triggered.

As deltas are computed and shifted into the correlation comparison register, they are also shifted into the

delta buffer and are held. Prefetch addresses are generated by accessing the delta buffer tail and proceeding toward the head (in LIFO order) until the desired prefetch degree has been reached. To compute the sequence of prefetch addresses the delta values are consecutively added to the miss address. The first two deltas in the delta buffer are ignored, because they are part of the current correlation. If the prefetch degree is larger than the number of entries in the delta buffer, then the delta buffer contains a complete recurring access pattern. In this case, the delta buffer is accessed more than once. After adding the last delta in the buffer (the head) to current miss address, the buffer's read pointer is reset to the tail of the buffer and the process continues. Figure 2 illustrates an example of an abstract CZone stride pattern, and shows the state of the GHB prefetching hardware when the pattern has been detected.

In addition to delta pair detection, the C/DC prefetcher also has a mechanism to quickly identify constant stride memory accesses. Before the second delta is shifted into the correlation key register, it is compared to the first delta (stored in the correlation key register during the previous cycle). If the first and second deltas match, a constant stride access pattern has been detected and addresses $a+s$, $a+2s$,…, $a+ds$ are prefetched.

3.2. Discussion

C/DC, like other recent prefetch research [10][12][16][18][19], was designed to prefetch into the lowest level cache (in our case the L2), because modern out-of-order processors can tolerate most L1 data cache misses with relatively little performance degradation. In this context, there are three main advantages to the C/DC prefetch approach.

First, localizing address streams based on CZones is well-suited for L2 prefetching because it does not require the program counter values of load instructions, but is still able to accurately prefetch regular access patterns.

Second, the use of a FIFO GHB naturally gives priority to recent program behavior and eliminates "stale" history by evicting the oldest miss address history. With the GHB, longer address streams are kept for the active miss streams, while the inactive ones age away. In contrast, a conventional prefetch table implementation would have a fixed length miss address stream in each table entry. Furthermore, the number of entries in a conventional table is chosen to reduce conflicts. Overall, the conventional approach often results in stale data and relatively large tables. Using CZones to index the GHB also reduces the number of bits stored in the GHB. With CZone prefetching, each GHB entry stores only the low-order bits of the miss address, because the high order bits are implied by the CZone tag that accesses the IT.

Third, the use of delta pair correlation is more general than using constant strides. Furthermore, delta pair correlation can significantly improve CZone prefetching when CZones are not completely effective at isolating address patterns. For example, consider an array of structs; each of which is 64 bytes (8, 8-byte words) in size. In this example words 0, 1, and 5 of each struct are accessed as part of a loop (each with a different load instruction). Assuming the array starts at memory location 0, the address stream and corresponding deltas are shown below.

Addresses	0	8	40	64	72	104	128
Deltas		*8*	*32*	*24*	*8*	*32*	*24*

If program counter values of the load instructions were available, then three constant stride patterns could be extracted from the address stream. However, in the absence of program counter values, simple strides do not appear, and a CZone method based only on strides would not be effective. On the other hand, there is a clear, detectable pattern in the delta values that C/DC can prefetch.

The apparent disadvantage of the GHB structure is the time required to walk the linked list and perform correlations. This delay is mitigated by the long latencies of cache misses to main memory – on the order of hundreds of cycles. Because of the long main memory latency, there is adequate time for accessing the GHB and performing correlations in a sequential manner. (Our simulator models the latency for all accesses to the GHB as the lists are walked.)

4. Evaluation of C/DC Prefetching

4.1. Simulation Method

A modified version of SimpleScalar 3.0 [3] was used for modeling an out-of-order processor. The processor configuration is summarized in Table 1. Memory bandwidth constraints are very important to data prefetching, therefore, we added a DDR memory model to SimpleScalar. The memory bus has two memory channels (See Figure 3), which is representative of modern high-end desktop systems. We assume average random access latency and row cycle time of 200 processor cycles. Cuppu et al. [7] have shown that average memory access latency on a realistic memory channel (e.g. less than 128 bits wide) is dominated by

bus access and data transfer time. Consequently, the simulator does not model internal DRAM operations (i.e. open rows, precharging, and refreshes).

Table 1: Processor configuration

Branch Predictor	Tournament Predictor 16K entry tables
Issue Width	4 instructions
RUU Size	64 entries
Load/Store Queue	32 entries
Level 1 D-Cache	8KB, 2-way set associative, 1 cycle latency, 32B lines
Level 1 I-Cache	16KB, direct mapped, 1 cycle latency, 32B lines
Level 2 Cache	512KB, 4-way set associative, 14 cycle latency, 64B lines

Each memory channel has a 16-entry miss status handling register (MSHR) queue for demand fetches and a 16-entry prefetch request queue. When the demand fetch MSHR queue reaches a high water mark, i.e. the point where outstanding cache accesses could fill the demand fetch MSHRs (if they all miss), instruction issue is forced to stall.

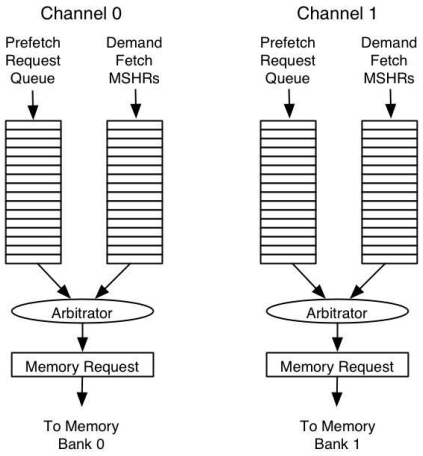

Figure 3: Memory Bus Interface

Demand fetches are given priority over prefetches; prefetches are allowed to access the memory bus only when the demand fetch MSHR queue is empty. If a prefetch request is made and the prefetch request queue is full, the head entry (the oldest prefetch request) is overwritten.

The SPEC CPU2000 benchmark suite is used to evaluate prefetch performance. However, benchmark *sixtract* is excluded because it intermittently generates unsupported system calls, which are very difficult to consistently reproduce and debug. All simulations skipped the first 2 billion instructions and simulated the next 2 billion instructions.

We compare C/DC with four recent prefetching methods designed to prefetch the miss address stream: CZone Prefetching [19], Distance Prefetching [15], Tag Correlation Prefetching [10], and Markov Prefetching [12].

The simulated prefetching methods inspect the L2 miss address stream and prefetch directly into the L2 cache. Before a prefetch is issued to the memory subsystem, the L2 cache's tag array is probed to ensure the prefetch address is not already in the cache. To keep prefetched (but not yet accessed) lines from modifying the "natural" L2 demand miss address stream, one bit prefetch tags are added to the L2 cache lines. When a prefetched line is written into the L2 cache, its prefetch tag is set. When a cache access hits a prefetched line with a set prefetch tag, the prefetch tag is cleared, and the access's memory address is sent to update the prefetch structures as if it were an L2 cache miss.

For each method, prefetch table configurations were chosen with two goals in mind: 1) each method should use a (near) optimal table configuration, and 2) when possible, all the methods should use approximately the same number of table entries. It is relatively easy to design satisfactory prefetch history tables for all methods except Markov and Tag Correlation Prefetching which have very different design constraints.

Originally, Markov Prefetching was proposed with an off-chip structure that prefetched the miss address stream of a small L1 data cache. In contrast, we consider prefetchers that are on-chip and prefetch the miss address stream of a large associative L2 cache. With this configuration, a 1MB correlation Markov table was insufficiently small, so we used a larger table (i.e. 8MB). Tag Correlation Prefetching performance is even more sensitive to the cache hierarchy and table sizes. We found that, in general, Tag Correlation Prefetching is less effective at prefetching the miss address stream of an associative cache, and the effect of conflicts in the pattern history table (PHT) is unpredictable. Hence, Tag Correlation Prefetching was also given a large table (i.e. 8MB), because its performance was better, in general, with a larger table.

The table configurations used throughout the rest of the paper are summarized in Table 2. Table sizes are rounded to the nearest kilobyte/megabyte. All tables are assumed to have a 1-cycle access latency (which gives a slight advantage to Tag Correlation Prefetching and Markov Prefetching).

Table 2: Prefetch Table Sizes

Prefetching Method	Notation	Table Configuration	Size
CZone Delta Correlation Prefetching	C/DC	256 IT entries 256 GHB entries	2 KB
CZone Prefetching	C/CS	256 table entries	6 KB
Distance Prefetching	G/DC	256 table entries	10 KB
Markov Prefetching	G/AC	256K table entries	8 MB
Tag Correlation Prefetching	CI/AC	2K THT entries 64K sets, 8 way PHT	8 MB

The notation given in the second column of the table follows a consistent taxonomy introduced in [18]. Each method is denoted as a pair: X/Y, where X is the key used for localizing the miss address stream and Y is the mechanism used for detecting addressing patterns. There are four localizing methods: CZones (C), Cache Indices (CI), and none – or global (G). And three detection mechanisms: Delta Correlation (DC), Address Correlation (AC), and Constant Stride (CS).

The baseline configuration for each method (except Tag Correlation Prefetching) has a prefetch degree of four. This is a reasonable level of aggressiveness for most methods and illustrates the methods' relative prefetch performance. Furthermore, a prefetch degree greater than four becomes unreasonable for Distance and Markov Prefetching, because increasing the prefetch degree beyond four does not improve performance and significantly increases both the correlation table size and memory traffic. Tag Correlation Prefetching does not include a prefetch degree capability (although a method to extend prefetch degree was suggested as future work [10]). For the baseline prefetch degree of four, we simulated the SPEC CPU2000 benchmark suite with different CZone sizes and chose a baseline CZone size (64KB) that was near optimal (averaged over the benchmarks).

4.2. Results

The benchmarks are divided into three groups. *Amiable* benchmarks are those where at least one prefetching method studied improves performance by more than 5%. All amiable benchmarks have less than 75% memory utilization without prefetching; this leaves sufficient memory bandwidth for prefetches. *Indifferent* benchmarks are those where none of the prefetching methods hurt performance, but no method improves performance by more than 5%. *Hostile* benchmarks are those where prefetching tends to degrade performance; this typically occurs when a benchmark has phases with very high memory utilization. The three benchmark groups are shown in Table 3. Figure 4 and Figure 5 show the IPC improvement and memory utilization for the amiable and hostile SPEC2000 benchmarks, the benchmarks where prefetching effects performance by at least 5%.

Table 3: Benchmark Groups

Amiable	Indifferent	Hostile
applu	apsi	ammp
facerec	art	mcf
galgel	equake	twolf
lucas	fma3d	vpr
mgrid	mesa	
swim	crafty	
wupwise	eon	
bzip2	gcc	
gap	gzip	
parser	perl	
	vortex	

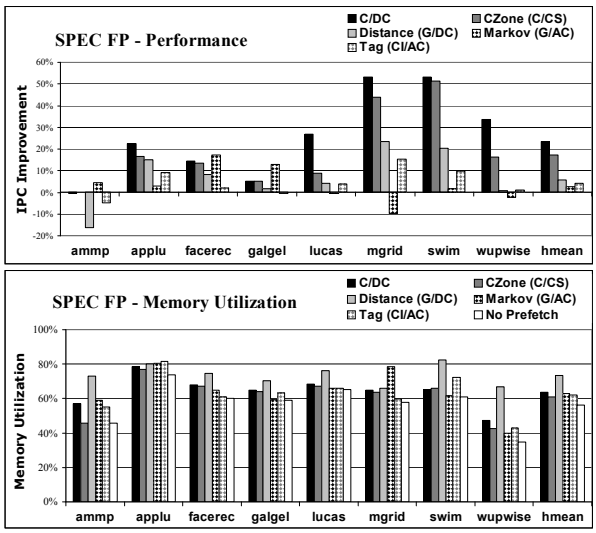

Figure 4: Spec FP IPC Improvement (top) and Memory Utilization (bottom).

C/DC performs as well or better than CZone prefetching (C/CS) on all amiable benchmarks, except for *bzip2*. On the floating point benchmarks, C/DC outperforms C/CS by as much as 15% (on *lucas* and *wupwise*), and by 7% on average. On the integer benchmarks, C/DC outperforms C/CS by as much as 6%; on average C/DC and C/CS perform the same.

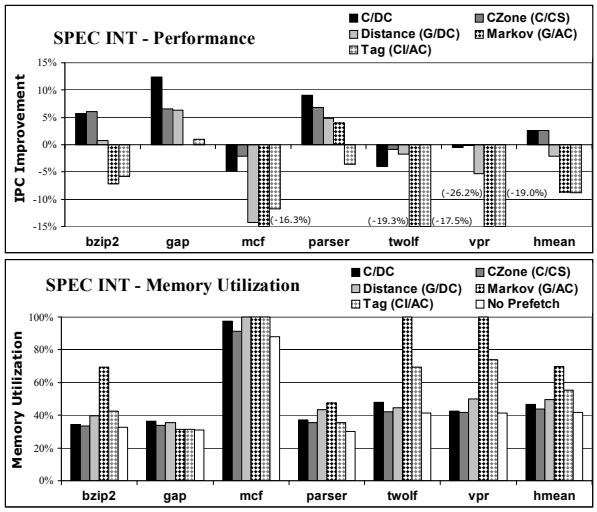

Figure 5: Spec INT IPC Improvement (top) and Memory Utilization (bottom).

For workloads that are sensitive to memory contention (i.e. the hostile ones: *ammp*, *mcf*, *vpr*, and *twolf*) inaccurate prefetches result in higher memory utilization that degrades performance of all the studied prefetchers. When studied with realistic memory constraints, as we have done, the additional memory traffic generated by prefetching is often detrimental to performance. Prior research on prefetch methods sometimes assumes much less constrained limits on available memory bandwidth (often unlimited). As will be described in Section 5, adding prefetch adaptivity to C/DC improves performance for the hostile benchmarks by turning off prefetching in situations where it is detrimental to performance.

Turning to the previously proposed correlation prefetch methods: Distance, Markov, and Tag Correlation prefetching (G/DC, G/AC, and CI/AC, respectively), we see that these methods usually under perform the CZone methods (C/DC and C/CS). The only exceptions are for benchmarks *ammp*, *galgel*, and *facerec*. In the case of *ammp*, *galgel*, and *facerec*, Markov prefetching (G/AC) performs the best (by a small amount), however, recall that Markov prefetching uses an 8MB table.

5. Adaptive Prefetching

Different programs use different data structures and access patterns. Consequently, the optimal CZone size and prefetch degree vary across programs. Even within a program, the data access patterns might change as the program goes through various phases of execution. Like most design parameters, the CZone size and prefetch degree can be optimized for good performance, *on average*, over a spectrum of benchmarks. However, this may lead to non-optimal performance (even performance losses) for certain benchmarks, which can be a deterrent to the use of more aggressive prefetch mechanisms.

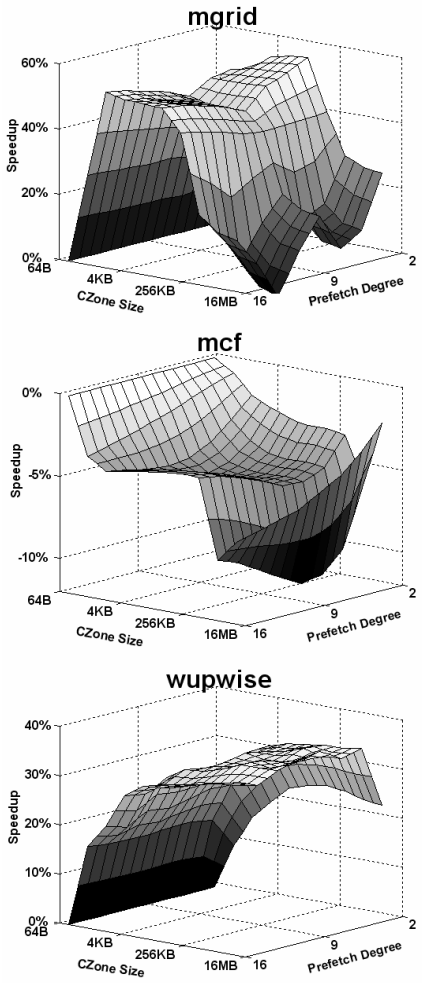

Figure 6: Performance variation with respect to CZone size and prefetch degree for benchmarks *mgrid*, *mcf*, and *wupwise*. Speedup is computed with respect to no prefetching.

Figure 6 shows the performance variation with CZone size and prefetch degree for three benchmarks. The surfaces are generated by simulating 81 configurations (ten CZone sizes times eight prefetch degrees, plus no-prefetching) and linearly interpolating between them. It is evident that these benchmarks have widely differing optimal configurations. *Wupwise* performs best with a large CZone size and a large prefetch degree, while *mgrid* works best with a moderate CZone size and small prefetch degree. In case of *mcf*, prefetching actually hurts performance, leading to as much as 11% performance loss with respect to using

no prefetching. Overall, performance is a strong function of the CZone size and varies moderately with the prefetch degree.

Figure 6 illustrates the opportunity for dynamically tuning the CZone size and prefetch degree to match a program (or program phase). Furthermore, dynamic tuning provides the opportunity to implement a *back-off* mechanism that turns off prefetching completely in those cases where prefetching hurts performance.

5.1. An Oracle Tuning Algorithm

In order to evaluate the performance potential of adaptively tuning the CZone size and prefetch degree, we first implemented an *oracle* tuning algorithm. The oracle algorithm divides program execution into fixed intervals of one million instructions. Performance is evaluated for a range of CZone sizes (from 64B to 16MB) and prefetch degrees (from two to sixteen). At the end of each interval, the algorithm compares performance of each of 81 configurations and selects the one that provides the best performance. Unnecessary reconfigurations are eliminated by reconfiguring only when IPC improves over the previous interval by at least 1%.

Due to limited simulation resources, we studied a proper subset of the SPEC CPU2000 benchmarks. The benchmarks were chosen to represent a mix of amiable (*facerec*, *lucas*, *mgrid*, *wupwise, and gap)* and hostile (*mcf* and *twolf*) benchmarks. We did not study indifferent benchmarks because, by our earlier evaluation, they neither benefit nor are hurt by prefetching.

Table 4: Performance improvement achieved with the oracle tuning algorithm versus baseline C/DC prefetching

Floating Point	Performance Improvement	Integer	Performance Improvement
facerec	8.7 %	gap	3.1 %
lucas	8.3 %	mcf	5.1 %
mgrid	4.0 %	twolf	4.2 %
wupwise	3.8 %		

Table 4 shows the performance improvement achieved by the oracle algorithm versus the baseline C/DC configuration used in Section 4, i.e. CZone size of 64KB and prefetch degree of 4. The oracle tuning algorithm is highly effective and can provide additional performance benefits of 2% to 9% beyond the baseline C/DC prefetch method. In hostile benchmarks such as *mcf* and *twolf*, tuning eliminates performance losses completely. Thus, it is evident that adding adaptivity has the potential for adding to prefetch performance gains. The next section describes a practical tuning algorithm that achieves a large fraction of the gains demonstrated by the oracle algorithm.

5.2. A Phased-Based Tuning Algorithm

Tuning algorithms have been proposed to control dynamically configurable hardware structures such as caches [2], branch predictors [14], and pipelines [1], among others. Recently, there have been proposals for tuning algorithms based on program phase detection [23][8][21][11]. These algorithms exploit the fact that programs go through phases of execution, with their behavior being relatively constant within a phase. Tuning is performed whenever a phase change is detected. The tuning algorithm we study (see Figure 7) uses instruction working set signatures [8][23] to detect program phase changes. A *working set signature* is a (lossy) compressed representation of the program's instruction working set. If the instruction working set signature changes significantly, it is assumed that the phase has changed.

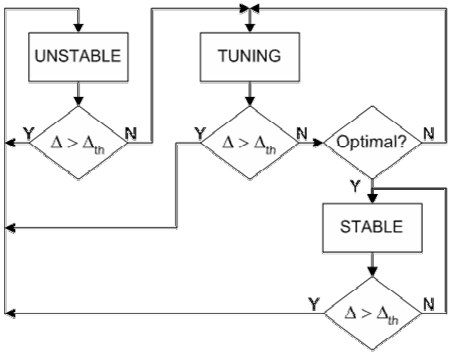

Figure 7: Tuning algorithm for dynamically adapting the prefetch mechanism. Δ represents the relative signature distance while Δ_{th} is the threshold.

The tuning algorithm begins in the UNSTABLE state with the configuration set to the baseline (CZone size 64KB, prefetch degree 4). At the end of each tuning interval (e.g. 1M instructions), the algorithm computes a relative working set signature change, or "distance" (Δ) with respect to the previous interval's working set signature, and compares it with a preset threshold distance (Δ_{th}) [8]. If the instruction working set change exceeds the threshold, a phase change is indicated and the algorithm stays in the UNSTABLE state; if not, the program is assumed to be stable and the algorithm transitions to TUNING. While in TUNING, the algorithm tries several different prefetch configurations. After all configurations have been tried, the algorithm chooses the one that provides the best per-

formance, and transitions to STABLE. If a phase change is detected while in the TUNING or STABLE state, it transitions to the UNSTABLE state and the configuration is reset.

The tuning algorithm can be implemented entirely in hardware or using a combination of hardware and software, for example with a co-designed virtual machine as proposed in [8]. The implementation requires a signature generation mechanism, a performance counter, and a control register for configuring the prefetcher. In the virtual machine implementation, the tuning algorithm is implemented in software. This leads to some performance overhead (< 0.1% for a 1M instruction tuning interval) but provides flexibility with respect to algorithm development, and possibly lower power dissipation. A hardware implementation suffers essentially no performance overhead because tuning decisions can be overlapped with computation, but it does require a small amount of additional hardware for storing the signature from the previous tuning interval, a mechanism for computing the relative signature distance [8], and a small state machine to control the algorithm. For evaluations here, we assume a hardware implementation.

5.3. Optimizations

5.3.1. Adapting Tuning Interval

Because program phases have different lengths and periodicities, a single tuning interval is not necessarily suitable for all programs. For example, a tuning interval of 1M instruction causes certain benchmarks (e.g. *facerec*) to spend a large fraction of time in the UNSTABLE state. Consequently, when tuning for a given program, it is best to first adapt the length of the tuning interval to the program. In this work, we choose either 100K or 1M instruction intervals; however, other intervals could be easily incorporated into the algorithm. For the first 250 million instructions simulated (before initiating tuning), the algorithm computes stability (i.e. fraction of time *not* spent in the UNSTABLE state) for tuning intervals of 100K and 1M instructions, and chooses the interval providing higher stability. When stability is similar, the larger tuning interval is given preference in order to minimize tuning overhead. For our evaluation the tuning interval is determined only at the beginning, but for long running programs, the algorithm for finding the best tuning interval can be initiated periodically to adapt to long term changes in program behavior.

5.3.2. Reducing Configuration Space

Tuning with 81 prefetching configurations, while useful for the oracle study, is probably too time-consuming in practice (unless the program phases are very long). Therefore, we systematically reduced our initial configuration space by making two observations based on Figure 6:

1. Performance at a given point is close to that at surrounding points. Therefore, choosing fewer evenly distributed CZone sizes and prefetch degrees should work reasonably well. We choose CZone sizes of 256B, 128KB, and 16MB, and prefetch degrees of 2, 8, and 16, in addition to the no-prefetching and baseline configurations.
2. The performance variation with CZone size is roughly the same for any given prefetch distance, and vice versa. Thus, the tuning of CZone size and prefetch degree can be done independently, e.g. first optimize CZone, then optimize prefetch degree – leading to a significant reduction in tuning combinations.

5.4. Evaluation

We evaluated the tuning algorithms by simulating each benchmark for 4 billion instructions. The simulator is fast forwarded by 1.75 billion instructions, after which the tuning algorithm searches for the best tuning interval for 250 million instructions. Performance is then measured over the next 2 billion instructions. Working set signature size is 128 bytes, and the threshold for detecting phase changes is set to 50% [8].

Figure 8 compares the Adaptive C/DC (AC/DC) prefetch mechanism against C/DC and C/CS. As discussed earlier, C/DC outperforms C/CS on most benchmarks, except the hostile ones. On hostile benchmarks, C/CS leads to smaller performance losses than C/DC mainly due to fewer prefetches generated. AC/DC, on the other hand, outperforms C/CS consistently on all benchmarks. On hostile benchmarks, AC/DC often turns off prefetching, thereby eliminating performance loss. On average, AC/DC improves performance by 10% over C/CS.

AC/DC also improves performance over C/DC on all benchmarks. In the case of amiable benchmarks, the performance improvements are a result of adapting the prefetch parameters (CZone size and Prefetch degree). In hostile benchmarks, the improvements are a result of turning off prefetching. On average, AC/DC provides 4% performance improvement over C/DC.

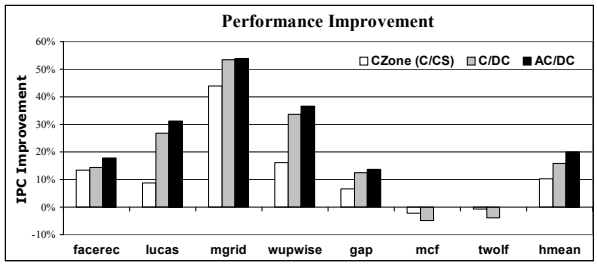

Figure 8: Speedups attained by the C/CS, C/DC and AC/DC prefetch mechanisms with respect to no prefetching.

6. Conclusions

We have proposed and studied innovative methods for data cache prefetching, aimed specifically at prefetching from main memory because modern out-of-order processors can tolerate most L1 data cache misses. The basic method, CZone / Delta Correlation prefetching (C/DC), combines CZone prefetching, GHB prefetching, and delta correlation, and has a number of advantages over existing prefetching methods.

- C/DC has the desirable property of not needing the program counter values of memory instructions causing misses, allowing the method to be easily implemented at lower levels of the memory hierarchy.
- C/DC has a very small prefetch table requirements (on the order of a few kilobytes), especially when compared to other correlation prefetching methods (sometimes requiring megabytes).
- Overall, C/DC outperforms the other prefetching methods studied. For example, C/DC outperforms conventional CZone prefetching by 6%.

Nevertheless, C/DC has some features that are undesirable. In general, performance of CZone prefetchers is sensitive to the CZone size, and optimal CZone size is a program dependent characteristic. Furthermore, C/DC, like the other correlation prefetching methods studied, is not as accurate as constant stride prefetching methods, which can lead to higher memory utilization and performance losses on benchmarks that are sensitive to memory contention.

We applied a phase-based adaptive tuning algorithm to solve the problems with C/DC and create a robust prefetching method that performs well on a diverse set of benchmarks. On average, the Adaptive CZone / Delta Correlation prefetching method (AC/DC) outperforms C/DC by 4%, and outperforms CZone prefetching (C/CS) by 10%. Adaptivity provides performance improvements not only by tuning CZone size and prefetch degree, but it also provides benefits by turning off prefetching in situations where performance is degraded.

Acknowledgements

This work was supported by NSF Grant CCR-0311361, by an Intel Graduate Fellowship, and by equipment donations and financial support from Intel Corporation.

References

[1] R. Bahar, and S. Manne, “Power and energy reduction via pipeline balancing,” in Proc. of the 28th Ann. Intl. Symp. on Computer Architecture, pp. 218-229, Jul. 2001.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general purpose architectures," in Proc. of the 33rd Ann. Intl. Symp. on Microarchitecture, pp. 245-257, Dec. 2000.

[3] D. Burger and T. Austin, “The SimpleScalar Toolset, Version 3.0.,” http://www.simplescalar.org.

[4] J. F. Cantin and M. D. Hill. Cache, “Performance for Selected SPEC CPU2000 Benchmarks,” Oct. 2001. http://www.cs.wisc.edu/multifacet/misc/spec2000cachedata/

[5] M. J. Charney and T. R. Puzak, “Prefetching and memory system behavior of the SPEC95 benchmark suite,” IBM Journal of Research and Development, 31(3), 1997.

[6] T. Chen and J. Baer, “Effective hardware based data prefetching for high-performance processors,” IEEE Transactions on Computers, 44(5), pp. 609–623, May 1995.

[7] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, “High-performance DRAMS in workstation environments,” IEEE Transactions on Computers, 50(11), pp. 1133-1153, Nov. 2001.

[8] A. S. Dhodapkar and J. E. Smith, “Managing Multi-Configuration Hardware via Dynamic Working Set Analysis,” In Proc. of the 29th Ann. Intl. Symp. on Computer Architecture, pp. 233-245, May 2002.

[9] J. W. C. Fu and J. H. Patel, “Stride directed prefetching in scalar processors,” In Proc. of the 25th Ann. Symp. on Microarchitecture, pp. 102-110, Nov. 1992.

[10] Z. Hu, M. Martonosi, S. Kaxiras, “TCP Tag Correlating Prefetchers,” In Proc. of the 9th Ann. Intl. Symp. on High Performance Computer Architecture, pp. 317-326, Feb. 2003.

[11] M. Huang, J. Renau, and J. Torrellas, “Positional adaptation of processors: application to energy reduction,” in

Proc. of the 30th Ann. Intl. Symp. on Computer Architecture, pp. 157-168, Jun. 2003.

[12] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," IEEE Transactions on Computers, 48(2), pp. 121–133, 1999.

[13] N. Jouppi, "Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers," In Proc. of the 17th Intl. Symp. on Computer Architecture, pp. 364-373, June 1990.

[14] T. Juan, S. Sanjeevan, and J. Navarro, "Dynamic history-length fitting: a third level of adaptivity for branch prediction," in Proc. of the 25th Ann. Intl. Symp. on Computer Architecture, pp. 155-166, June 1998.

[15] G. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB Prefetching: an application-driven study," In Proc. of the 29th Ann. Intl. Symp. on Computer Architecture, pp. 195-206, May 2002.

[16] S. Kim and A. Veidenbaum, "Stride-directed prefetching for secondary caches," In Proc. of the 1997 Intl. Conf. on Parallel Processing, pp. 314-321, Aug. 1997.

[17] A. Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," In Proc. of the 28th Ann. Intl. Symp. on Computer Architecture, pp. 144-154, July 2001.

[18] K. Nesbit, and J. E. Smith, "Prefetching with a global history buffer," In Proc. of the 10th Ann.l Intl. Symp. on High Performance Computer Architecture, pp. 96-105, Feb. 2004.

[19] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," In Proc. of the 21st Ann. Intl. Symp. on Computer Architecture, pp. 24-33, May 1994.

[20] Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam, "MicroLib: a case for the quantitative comparison of micro-architecture mechanisms", Workshop on Duplicating, Deconstructing, and Debunking, Munich Germany, June 2004.

[21] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in Proc. of the 30th Ann. Intl. Symp. on Computer Architecture, pp. 336-347, Jun. 2003.

[22] A. J. Smith, "Sequential program prefetching in memory hierarchies," IEEE Computer, 11(12), pp. 7-21, Dec. 1978.

[23] J. E. Smith, and A. S. Dhodapkar, "Dynamic microarchitecture adaptation via co-designed virtual machines," in 2002 Intl. Solid State Circuits Conf., pp. 198-199, Feb. 2002.

[24] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," In Proc. of the 29th Ann. Intl. Symp. on Computer Architecture, pp. 171-182, May 2002.

[25] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," IBM Technical White Paper, 2001.

[26] S. VanderWiel and D. Lilja, "Data prefetch mechanisms," ACM Computing Surveys, 32(2), pp. 174–199, June 1999.

[27] Z. Wang, D. Burger, K. McKinley, S. Reinhardt, C. Weems, "Guided region prefetching: a cooperative hardware/software approach," In Proc. of the 30th Ann. Intl. Symp. on Computer Architecture, pp. 388-398, June 2003.