2011

Potential for hardware-based techniques for reuse distance analysis

Justin R. Slepak

Michigan Technological University

THE POTENTIAL FOR HARDWARE-BASED TECHNIQUES FOR REUSE DISTANCE ANALYSIS

By

Justin R. Slepak

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

(Computer Science)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2011

This report, "The Potential for Hardware-Based Techniques for Reuse Distance Analysis," is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE

Department of Computer Science

Signatures:

Co-Advisor _____
Dr. Steven M. Carr

Co-Advisor _____
Dr. Zhenlin Wang

Committee Member _____
Dr. Donald L. Kreher

Department Chair _____
Dr. Steven M. Carr

Date _____

Contents

List of Figures

Abstract

Reuse distance analysis, the prediction of how many distinct memory addresses will be accessed between two accesses to a given address, has been established as a useful technique in profile-based compiler optimization, but the cost of collecting the memory reuse profile has been prohibitive for some applications. In this report, we propose using the hardware monitoring facilities available in existing CPUs to gather an approximate reuse distance profile. The difficulties associated with this monitoring technique are discussed, most importantly that there is no obvious link between the reuse profile produced by hardware monitoring and the actual reuse behavior. Potential applications which would be made viable by a reliable hardware-based reuse distance analysis are identified.

Chapter 1

Introduction

1.1 Background

Despite regular increases in processor speed over time, memory accesses have not kept up with this trend. Because memory operations form a significant proportion of a typical program, the resulting speed disparity makes memory acces a serious performance bottleneck. Techniques used for mitigating slow memory response require awareness of the CPU architecture and the runtime behavior of a program. CPU caches hold recently-used segments of data in the expectation that those segments will be accessed again soon. Because the cache is smaller than RAM and located on the CPU chip itself, the time needed to access data in the cache is reduced compared to RAM. Holding recently-used data exploits two well-known tendencies of typical programs: that individual data elements are

often used once and then used again shortly afterwards ("temporal locality") and that data elements near those recently used are likely to be used in the near future ("spatial locality"). Instruction scheduling by the compiler allows load instructions to be issued as early as possible, increasing the chance that the load will complete before the resulting data is needed (otherwise, the CPU will have to stall execution while it waits for the data). Prefetching requests data from memory in anticipation of future use. It can be done by the compiler ("software prefetching") by inserting special "prefetch" instructions into the compiled program or by the processor ("hardware prefetching") using various techniques which analyze memory usage patterns to predict near-future data use. Load speculation is the reordering of program instructions so that a load instruction is issued before a store instruction which precedes it in program order. Again, issuing the load earlier allows more time for it to complete, but a recovery mechanism is needed in case it turns out that the speculated load depends on the result of the prior store (i.e. they operate on the same memory address).

1.2 Motivation for Hardware-based Reuse Distance Analysis

For a pair of memory accesses to the same location (a "reuse pair"), the "reuse distance" is defined as the number of distinct memory locations accessed between that reuse pair. Because the cache operates by keeping recently-used data readily accessible, reuse distance identifies what data elements will be kept in or evicted from the cache before being reused; thus a program's reuse distance profile gives a metric for the temporal locality

of that program. Tracking a reuse distance profile for each individual memory instruction gives additional insight useful for the latency mitigation techniques described above. Knowing approximately when a data element will be reused helps decide whether that piece of data is worth keeping in the cache (n.b. perfect caching requires evicting the data whose next use is farthest in the future, whereas the conventional "least recently used" policy is an approximation to this), when to schedule a load or prefetch instruction (or issue a hardware prefetch), or whether a load is likely to depend on a particular store. Some memory use optimizations require identifying a program's critical instructions, which are those responsible for most (typically 90%) of the program's cache misses.

Despite algorithmic improvements such as replacing the LRU stack of Mattson et al. [10], a stack-like structure holding memory references ordered by recency of use so that stack depth is equivalent to reuse distance, with a tree-based structure [5], current techniques for gathering reuse distance profiles run very slowly. Data is gathered via software instrumentation, and an instrumented program runs much slower than the original version of that program. In practice, this means analysis must use training runs on smaller input data. The technique proposed herein uses hardware performance monitors rather than instrumentation. The monitored program runs with little reduction in speed, and information is accessible at runtime. This makes it possible to perform optimization at runtime, such as by a JIT compiler, and to perform analysis on actual program input data rather than smaller training inputs.

1.3 Outcome

The hardware-based measurement technique proposed herein produces similar per-instruction output for a wide range of tested programs. In most cases, there is little variation between the output for an individual benchmark's critical instructions. Most of the variation is across different benchmark programs, but one histogram shape appears in over half of the tested programs.

Chapter 2

Related Work

This chapter describes prior research on which this report is based. The work comes primarily from two areas, reuse distance analysis and hardware performance monitoring. Much of the work deals with analysis of program cache usage for profile-based optimization purposes.

The reuse distance for a consecutive pair of memory references to the same location is defined as the number of unique memory locations accessed between the references in question. For example, given a memory access trace (A, E, C, D, B, A, B, C, E), the reuse distance between the two accesses to A is 4, while the reuse distance between accesses to C is 3. Reuse distance gives a description of a program's temporal locality; a cache using LRU replacement will have available those memory locations corresponding to short reuse

distances. This allows reuse distance to be compared with cache capacity to identify data which will be evicted before reuse, thereby leading to prediction of cache misses. This prediction does not take into account conflict misses, but Fang et al. [6] find that they are not common. It also ignores compulsory misses, but those are exactly the cache misses where data reuse has not yet happened.

Processors typically include facilities for monitoring their own performance. This includes specialized registers for counting performance-related events (such as retired instructions, cache misses, etc.) as well as, in more recent processors, the capability to save data associated with an event (e.g. the address of the instruction which caused the event) into a buffer in memory. Use of hardware performance monitoring allows profile-based optimization with far less overhead than that associated with simulation- or instrumentation-based systems. Several authors describe techniques for monitoring via hardware and for making use of the resulting data.

2.1 Reuse Distance Analysis

Memory references can be tracked in a stack-like structure, with new references being added to the top and old references moved from the middle to the top when they reappear in a program's memory trace. The depth of a memory reference in the stack at the time it is moved back to the top is the reuse distance between the current and prior references to that

address. However, this requires storing a history of the most recent access to each memory location and searching linearly on each memory access, with a total space cost in $O(m)$ and time cost in $O(m*n)$, where m is the number of distinct memory locations accessed, and n is the number of memory accesses made during runtime. This makes monitoring unfeasibly slow on anything more than a small memory access trace. Ding and Zhong [5] and Zhong and Chang [12] both describe techniques for reducing the performance overhead associated with reuse distance monitoring.

Ding and Zhong propose a tree-based structure for tracking reuse distances, in which long distances are approximated within a relative linear error bound, reducing the space cost to $O(\log m)$ and the time cost to $O(n*\log\log m)$. A modification which respects a constant error bound is also discussed. Because a full program run's access trace can now be converted to a reuse distance trace, two training runs on small data sets are used, and a histogram of reuse distances for each run is constructed. This *reference histogram* is built by sorting all memory accesses by reuse distance and then splitting the sorted list into equally-sized bins. The value for each bin is the average reuse distance for memory accesses in that bin. For each histogram bin, the training runs are used to model the histogram bin's value as an affine function of *data size*, which is defined as the largest single reuse distance in the memory access trace (this is, in effect, the maximum size of the working set of program data). Data size itself is estimated by sampling reuse distances which are greater than a chosen threshold distance. From the sequence of above-threshold reuse distances, the analyzer selects the first k local maxima which cover at least m data samples (in practice, $k = 1$

and $m = 2$ were found to be sufficient for most of the programs tested). The largest of these peaks is selected as the data size estimate; while it is really only linearly proportional to the data size, this is enough for the linear fitting process described above (the actual factor by which it differs depends on program structure). More discriminate methods are suggested for selecting the peak, but they are not put to use.

Zhong and Chang [12] gather only partial segments of the full program's memory reuse trace using a sampling method based on bursty tracing. The sampling system is turned on and off according to a timer or counter, thus profiling discrete sections of a program run. This distinguishes it from the PEBS-based technique proposed herein, which simply selects individual instructions from a run rather than sequences of instructions. Naïve sampling directly applies bursty tracing with preplanned "sampling" and "hibernating" intervals. By sampling $\frac{1}{r}$ of the execution, sampling overhead is reduced by a factor of r, but there is no guarantee of accuracy in measurement. Only reuses confined to a single sampling interval can be measured accurately, and this technique allows a very high error rate to go undetected. In biased sampling, the monitor still searches for the last access time of memory references encountered during hibernating intervals. If it is found that the previous access was during the same hibernating interval, the access is ignored. Otherwise, an access-time table and access trace must be updated. If there was a prior occurrence of this reference before this hibernating interval, its reuse distance is calculated and included in the histogram. Under this technique, the measured reuse distance is always greater than or equal to the actual reuse distance because of the references which are ignored during

hibernation. Not all distances have an equal chance of being sampled, because those not long enough to reach before the current hibernating interval are ignored, while reuses with distances longer than the hibernating interval are guaranteed to be included in the sample. The final revision of the sampling technique is "history-preserved representative sampling," in which bookkeeping is performed during sampling and hibernating intervals. Only those reuse pairs whose first element is in a sampling interval are measured for distance, equalizing the probability of inclusion in the sample for all possible reuse distances. Several additional performance improvements for this algorithm are described as well. Using this algorithm, the authors sample 1% of reuses and achieve an average speedup of 7.5 over Ding and Zhong's non-sampling algorithm.

Fang et al. [6] apply reuse distance analysis to individual instructions, again to predict reuse distances based on training runs with smaller input data. The reuses are associated with the instructions which caused them, generating a reuse histogram for every static memory access instruction. A histogram is then split into access patterns, meant to correspond the distinct behavioral patterns a single instruction may show. Each histogram pattern is a region around a local maximum, bounded by local minima or zeroes. A pattern is described with two linear functions, one from left minimum to maximum, and one from maximum to right minimum. The predictive analysis focuses on mapping a program's data size to an individual instruction's pattern arrangement. For critical instructions, increasing data size typically shifts one or more patterns to the right, preserving the general distribution of reuse distances within each single pattern. Software instrumentation is used on two training runs,

similar to the technique used by Ding and Zhong. Most tested benchmarks showed over 90% prediction coverage and accuracy, but prediction is limited when the training runs do not achieve full code coverage.

In further study, Fang et al. [7] consider prediction of cache misses based on reuse distance analysis. The cache size represents a cutoff point in the reuse distance histogram: bins to the right of the cutoff represent cache misses due to reuse distance longer than cache capacity, while those to the left are cache hits. While conflict misses are not taken into account in this model, they turn out to be quite rare. The predicted miss count, the sum of histogram bars right of the cutoff, is used to identify critical instructions, i.e. those responsible for most of the program's cache misses. These can also be identified using a reuse histogram predicted based on training runs, a technique referred to as "predicted reuse distance." This was compared against "reference reuse distance," the predicted miss count calculated from a histogram generated by running the larger reference-data program run under instrumentation, representing an upper bound on what can be learned about cache miss rate via reuse distance analysis. It was also compared against "test cache simulation," measuring cache miss rates via a cache simulation on the smaller of the training data sets. The test cache simulation was consistently beaten by the predicted reuse distance technique.

Keramidas et al. [8] apply reuse analysis to L2 cache management. The processor keeps a table mapping program counters to predicted reuse times and confidence values. The predicton is represented as the log of the expected reuse time, and the confidence value is

tracked as a 2-bit saturating counter. When a line is to be evicted, two candidates are identified. The first is the line whose predicted next access time is farthest in the future; this candidate is selected to maximise $score = (predictedreusetime - timesincelastaccess)$. If a line has a negative estimated time of access, its time is clipped to 0. The second candidate is selected according to LRU policy, with *score* equal to the time it has been in the cache. The candidate with the higher score is evicted. The confidence score is incremented when actual reuse time matches the prediction and decremented on a mismatch. When a confidence score is reduced to 0, the old prediction is replaced with the most recent actual time. A table tracking 512 instructions with 39 bits per entry (32 for program counter, 5 for predicted log, 2 for confidence counter) was found to be sufficient to capture most of the benefit offered by this technique. Only a few instructions are traced at a time, limiting the number of watchpoint registers. Keramidas et al. [9] found that this technique is advantageous in comparison to shepherd caching and dynamic insertion policy, proposed by other authors, because they do not take miss cost into account, and that it is superior to prior work on replacement based on awareness of memory-level parallelism due to its capability of handling LRU-hostile cache access behavior, which is commonly seen at the L2 cache.

The work by Keramidas differs from the technique proposed here in its characterization of reuse behavior according to time rather than number of unique intervening memory references and in its selection of individual instructions to track rather than sampling from the entire memory access trace. Measuring reuse distance in hardware would require more than just a single register per tracked reuse, as counting only unique intervening memory

references requires remembering which ones have already occured.

2.2 Hardware Performance Monitoring

Buck and Hollingsworth [2, 3] consider sampling L2 cache misses for profile-guided optimization. A miss count is associated with each dynamically-allocated memory block. A system using separate cache miss counters for different sections of memory is also considered, but this proposal assumes hardware support for such counters (though they could be approximated with PEBS-based monitoring). Both techniques are usable for identifying what memory structures are responsible for the most cache misses. Early work used instrumentation to identify the addresses responsible for cache misses.

Later work used an Itanium 2 processor which includes support for hardware-based sampling of cache misses and makes their proposed sampling technique possible via hardware performance monitoring. Load accesses to the L2 cache (i.e. floating point loads and L1 load misses) are sampled, and each miss address is associated with source-level data structure. The authors make a detailed examination of two SPEC CPU2000 benchmarks with low L1 cache hit rates, `equake` and `twolf`. In equake, only 64% of loads hit the L1 cache, with most of the high-latency loads coming from iterating over a single 3-dimensional array of doubles. Changing the array allocation to one contiguous block of memory rather than using a separate `malloc` call at each indirection level resulted in a 57% decrease in

L1 misses, a 30% decrease in L2 misses, and a 10% decrease in running time. `twolf` had a cache hit rate of 74%, with a slightly higher average memory access latency than `equake`. Most cache misses were attributed to a small handful of C struct types. Some were small enough to fit within a single L1 cache line, and all were small enough to fit in an L2 cache line. However, it is unlikely that multiple entire structures fit into a cache line due to `malloc`'s space overhead. By writing a specialized memory allocator for these small structures, they could be placed contiguously in memory. The custom allocator also accepts hints from the caller about which structs include pointers to each other and co-locates such structs. This optimization eliminated 57% of L1 misses and 27% of L2 misses, reducing run time by 11%. A separate optimization focused on a variable-sized 2-dimensional array, which was the next most frequent cause of L1 misses. Allocating the array in a contiguous block, as was done in `equake` reduced L1 misses by 33%, L2 misses by 29%, and running time by 16% versus the original `twolf` program.

This cache miss sampling technique is distinguishable from the technique proposed in this report because it does not attempt reuse distance analysis, instead identifying the source-level data structures responsible for the majority of cache misses. The optimizations performed based on information thus gathered also involved extensive human intervention, as the critical steps (providing structure names to the monitoring tool, identifying problematic sections of source code, and transforming the source code) were all done manually.

Schneider et al. [11] and Cuthbertson et al. [4] demonstrate the use of hardware perfor-

mance monitoring in a just-in-time (JIT) compiler. Schneider et al. used a Java Virtual Machine (JVM) with a copying garbage collector designed to improve locality by co-locating objects on the heap. The efficacy of co-location can be guided in advance and checked afterwards by sampling cache misses via PEBS. Cuthbertson et al. used an Itanium processor's performance monitoring unit, which supplies the addresses of the instruction and memory data associated with a sampled cache miss, as well as the latency of the memory access. A JVM internally tracks the latencies associated with bytecode instructions and maps them to specific segments of the JIT compiler's internal representation of the code. The resulting load latency profile is used for optimizing JIT compilation. Global instruction scheduling attempts to take advantage of code motion opportunities. While it is common to minimize the effect of memory latency by scheduling memory operations as early as possible, this JVM is modified to deemphasize the early scheduling of low-latency loads, thus shortening register live ranges and reducing the amount of data which must be spilled to memory. Cuthbertson et al. also used an object co-locating garbage collector similar to the technique described by Schneider et al.

Chapter 3

Data Collection Method

Experiments were run on an Intel Xeon processor designed around the Intel Core architecture. This is a four-core CPU with two 4 MB L2 caches, but all tested workloads are single-threaded, and a single core can only make use of one L2 cache. SPEC CPU2006 benchmarks were run on Fedora 8 with the perfmon kernel modification.

The proposed monitoring technique is based on Intel's Precise Event-Based Sampling ("PEBS"), a feature now included in several of Intel's processor lines. PEBS allows a particular performance-related event to be specified (in this case, L2 cache misses) for sampling, as well as the sampling interval. When a sample is collected, the data includes the instruction pointer (`%eip` in x86) of the instruction following the one responsible for the event and the contents of the architectural integer registers when the event happened. In

most cases, this is sufficient information to identify the responsible instruction and reconstruct the memory address it accessed. However, this analysis is difficult across function calls or even basic block boundaries; in this study such samples are ignored. PEBS has limitations in that it only works on a small subset of performance events, and it may interfere with some uses of performance counter registers (e.g. it is not possible to both sample L2 cache misses and count them).

The perfmon kernel extension and user-level library [1] provide an interface to hardware performance monitoring unit (PMU) with the intent of abstracting some of the differences between PMUs on various architectures and CPU models. In order to use perfmon for monitoring a program's execution, a separate monitor process is created and attached to the program under test. The monitor program identifies a particular set of events to count and/or sample. At most one event may be chosen for sampling, and some selections of events to count/sample may be rejected by perfmon as unavailable. In this study, the monitor program samples L2 cache misses, and it counts L1 cache misses to estimate the number of unique memory references between consecutive L2 cache miss samples. Because the PEBS-based measurement tends to produce very long reuse distances, in the output histograms shown here, the resulting count is divided by 1000 to shift the histogram to the left.

The Pin instrumentation system was used for comparison. The tool used is a small modification to that used by Fang et al. in [7], in which each instruction is associated with

the distribution of its reuse distances among a series of histogram bins. While Fang et al. used bins growing logarithmically in size from 1 to 1k, followed by bins of size 1k (up to a maximum measurable distance of 64k), this technique focuses attention on long reuse distances. As such, the logarithmic-linear bins are replaced with purely logarithmic bins, with the maximum measurable distance increased to 2^{30}. Because this tool works by direct measurement of program behavior, the histograms shown as Pin results are the actual reuse profiles for the given instructions.

Chapter 4

Data Summary

In this chapter, PEBS and Pin histograms collected from 26 SPEC CPU 2006 benchmark programs are described with a focus on looking for relationship between output of the two collection methods. While a typical program may only show one to three visually distinct shapes in its PEBS histograms, there will usually be a wider range of Pin histogram shapes. Some programs may have an unusually wide variety of PEBS shapes or an unusually narrow range of Pin shapes, but the general shape of an instruction's PEBS histogram is generally not enough to determine the general shape of its Pin histogram.

4.1 Per-Benchmark Examination

4.1.1 Overview

For most benchmarks, typical Pin and PEBS histograms for critical instructions are shown. For a few specific benchmarks, especially those with cases of odd behavior or particularly small critical sets, resulting data is discussed in more detail.

The most common PEBS histogram shape is made up of a single large pattern, usually peaking the 5th bin, and tapering off to the left. Examples of this shape are generated by the majority of these benchmarks (most benchmarks for which no PEBS examples are shown only produce this shape), and its commonness presents an obstacle to analysis based on this sampling technique.

4.1.2 400.perlbench

While this program's critical instructions show more variation than normal in their Pin histograms (the top ten instructions alone show six distinct histogram shapes), most of the PEBS histograms (including nine of the top ten) have the same shape as seen in figure 4.1.

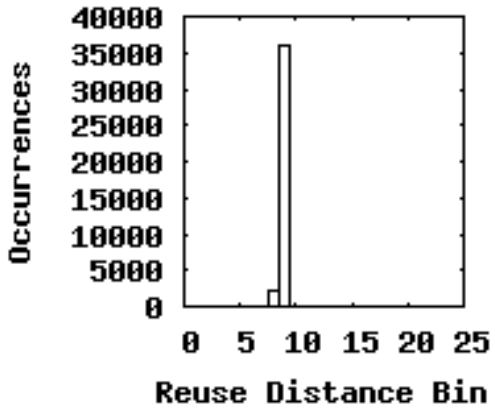

Figure 4.1: 400.perlbench typical PEBS histogram

4.1.3 401.bzip

The three common Pin histogram shapes are shown in figure 4.2. All three example instructions produce PEBS histograms with the same shape, as in figure 4.3

4.1.4 403.gcc

Most of the critical instructions in this program have PEBS histograms with multiple patterns, and there is significant variation among their shapes, suggesting that 403.gcc may be a good target for a PEBS-based reuse distance prediction technique.

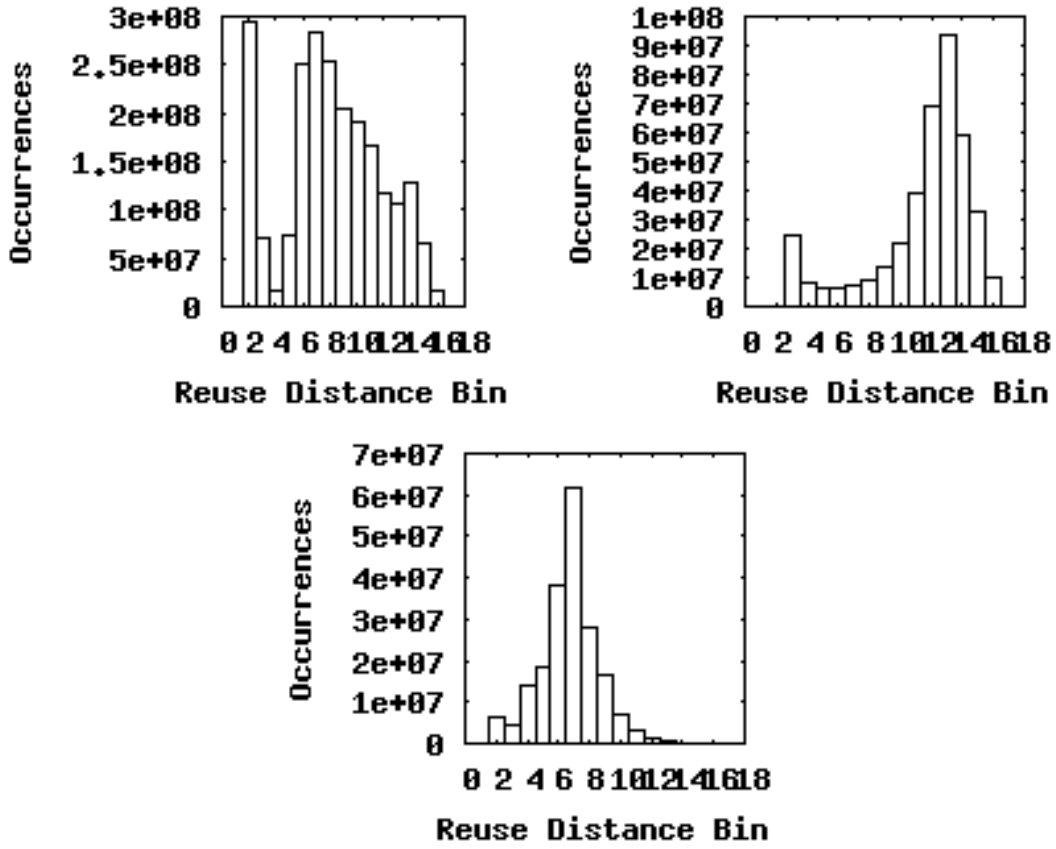

Figure 4.2: 401.bzip2 typical Pin histograms

4.1.5 410.bwaves

This benchmark has two recurring Pin histogram shapes. One appears in 7 of the 21 critical instructions and covers 71.2% of the cache misses; the other appears in 5 critical instructions and covers only 6.3% of cache misses. These patterns are shown in figure 4.4. The program shows a nearly uniform shape in PEBS histograms, with minor variation on the left side of the main pattern (see figure 4.5).

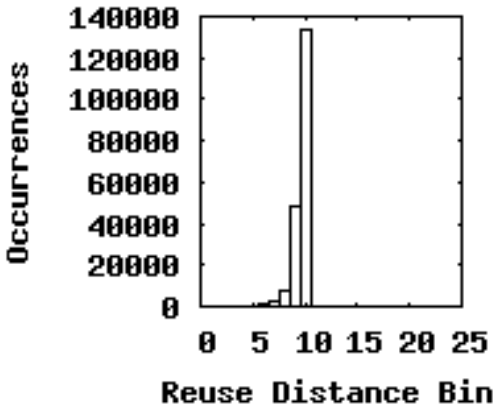

Figure 4.3: 401.bzip2 typical PEBS histogram

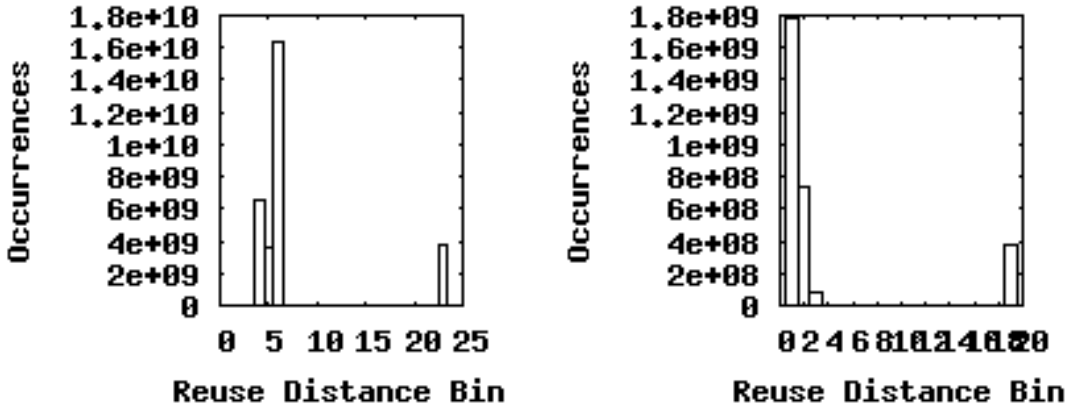

Figure 4.4: 410.bwaves Pin histogram shapes

4.1.6 429.mcf

Most PEBS histograms here have a single major pattern centered at very low distance. The particular variation in which this pattern is a single column (with a smaller, disconnected column at 0) appears in 14 of the 25 critical instructions, which account for 75.2% of cache misses (an example is given in figure 4.6). Pin histograms have several groups of two or three instructions sharing the same general shape, but there is much more variety among

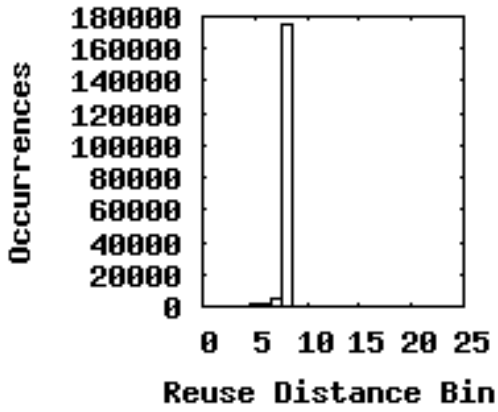

Figure 4.5: 410.bwaves PEBS histogram shape

the shapes to reduce it to a few common cases.

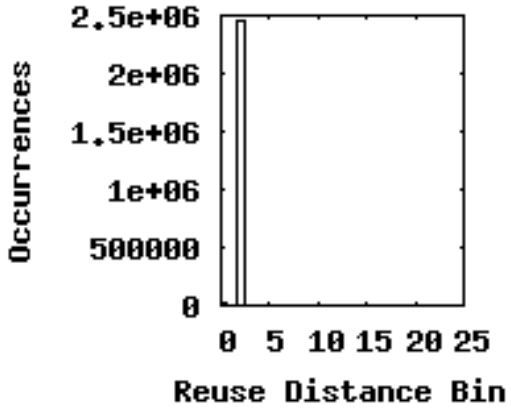

Figure 4.6: 429.mcf PEBS histogram shape

4.1.7 434.zeusmp

Most Pin histograms show two or three well-separated groups, each typically composed of one or two patterns. The left-most group is typically the largest and narrowest (often a single column). Most PEBS histograms have one major pattern and some shorter-distance

noise. The instruction responsible for the largest number of cache misses gives a good example (see figure 4.7).

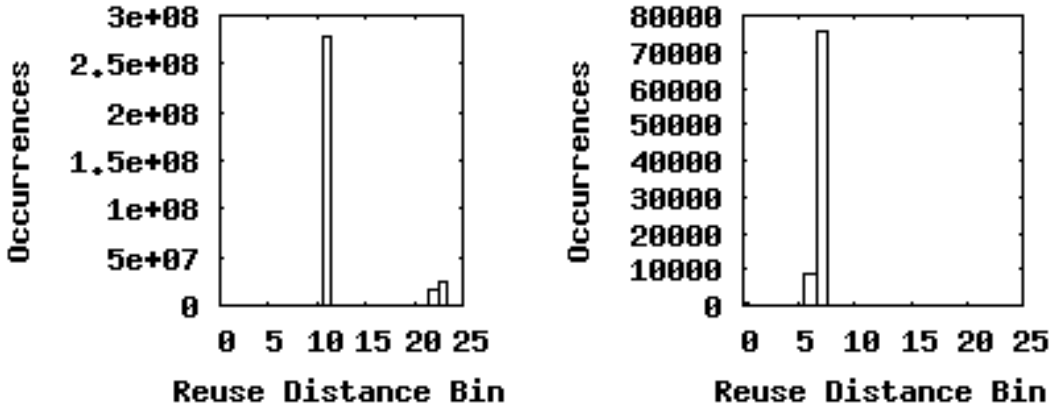

Figure 4.7: 434.zeusmp typical Pin histogram (left) and PEBS histogram (right)

4.1.8 435.gromacs

Pin histograms in this benchmark have several shapes, but the one shown in figure 4.8 is the only common shape, appearing in 7 of 18 instructions' histograms (covering 68.6% of cache misses). All PEBS histograms have the same general shape, a single group with one large pattern (the main spike) and a fairly flat (and near-zero) range to the left (see figure 4.8).

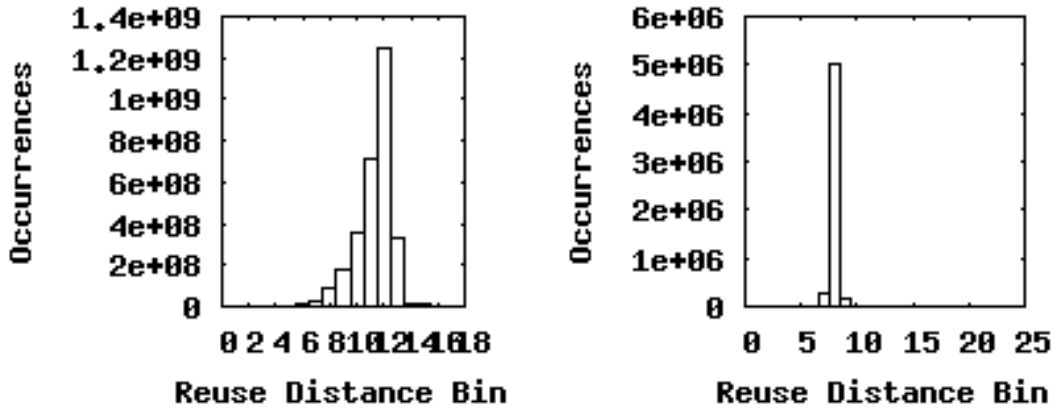

Figure 4.8: 435.gromacs typical Pin histogram (left) and PEBS histogram (right)

4.1.9 436.cactusADM

This is the first of three benchmarks with unusually large critical sets. In this benchmark, 27.3% of the sampled instructions are needed to cover 90% of the sampled misses (compared with a mean of 11.1% needed). However, two thirds of the sampled instructions, including all of the critical instructions, come from a single function, `Bench_StaggeredLeapFrog2`, which is the core of the numerical algorithm used by the program to solve a coupled system of nonlinear partial differential equations.

The pin histograms for the critical instructions almost all have two narrow patterns with wide separation between them. Several instructions have narrower spacing between the two patterns, but these instructions' PEBS histograms are not shaped differently from those of other instructions. Typical examples are shown in figure 4.9.

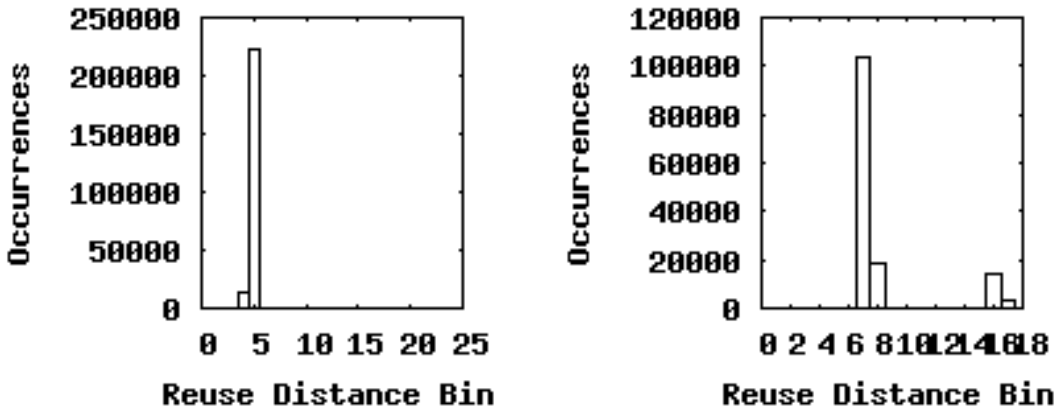

Figure 4.9: 436.cactusADM typical PEBS histogram (left) and associated Pin histogram (right)

4.1.10 444.namd

As in several other benchmarks, the typical PEBS histogram is made of two disconnected patterns, with the rightmost one far larger than the other. At the standard bin resolution, some instructions appear to give an upper pattern weighted fairly evenly or even towards lower reuse distances, but at finer resolution, this is corrected (see figure 4.10 for comparison). The appearance of reversed weighting may also be eliminated in the bin-merging process used by Fang et al. [6], though because the resulting single bin is assumed to be uniformly distributed, this is a change to a less wrong conclusion. Pin histograms show three common patterns, shown in figure 4.11.

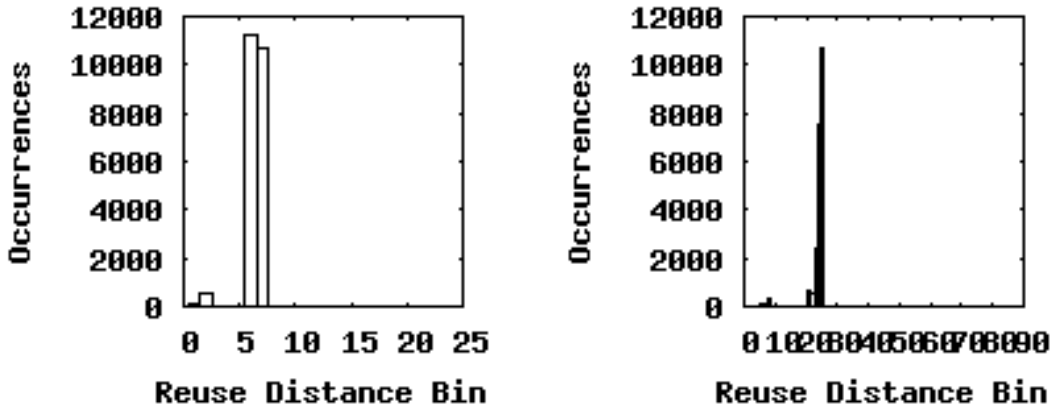

Figure 4.10: This 444.namd PEBS histogram suggests the wrong interpretation at coarser resolution. Bins on the left grow by a factor of 2, while those on the right grow by a factor of $\sqrt[4]{2}$

4.1.11 445.gobmk

This benchmark produces PEBS histograms with a single major pattern. It has a slower taper on the left than on the right, and the extension of the left taper is the primary difference in the shape of these histograms. Figure 4.12 shows a typical example. A wide variety of Pin histogram shapes appear among the critical instructions.

4.1.12 447.dealII

Again, the typical instruction's PEBS histogram includes only one significant pattern, with the histograms themselves differentiated primarily in the histogram's left side taper. However, the patterns for this benchmark tend to be a bit narrower. This benchmark also shows almost as many different pin shapes as 445.gobmk despite having a much smaller critical

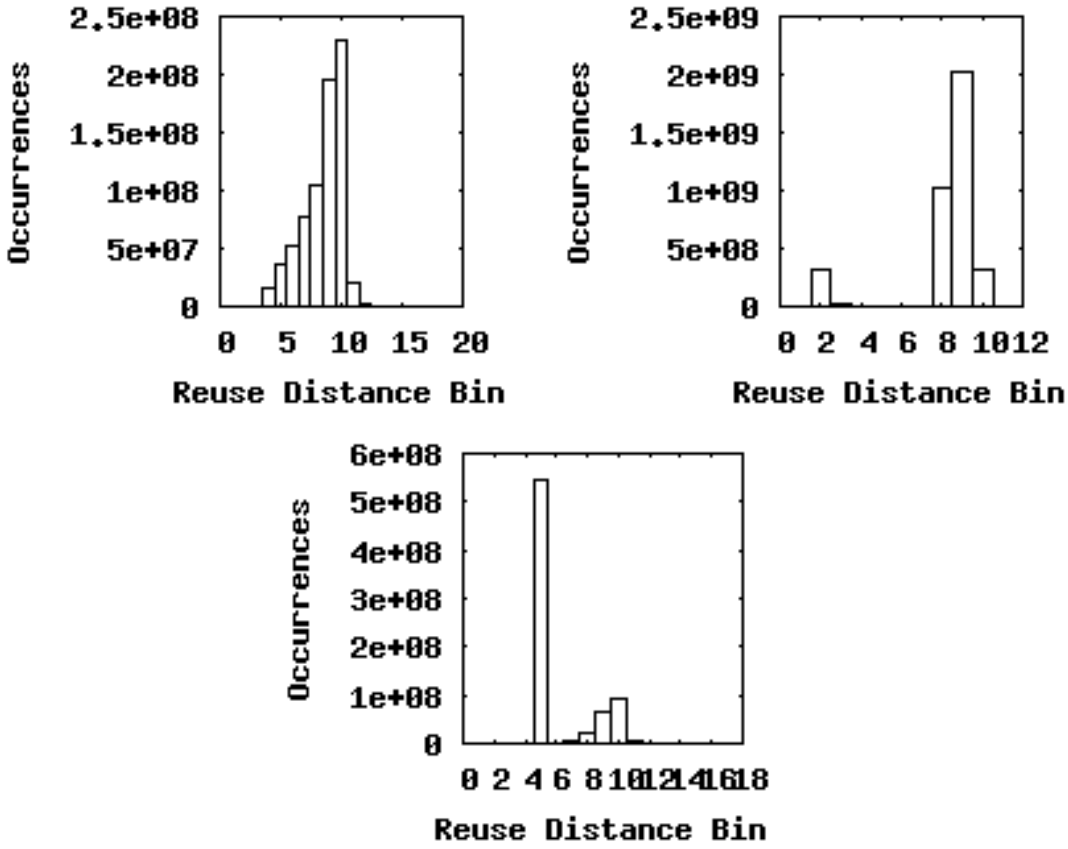

Figure 4.11: 444.namd typical Pin histogram shapes

set.

4.1.13 450.soplex

Most PEBS histograms in this benchmark are similar to those in 444.namd, but 13 of the 83 critical instructions, including the top two miss-causing instructions, have a wider pattern (though centered around the same bin). These wider instructions account for 41.4% of the cache misses. One instruction comes from the `SSVector::setup()` function, one

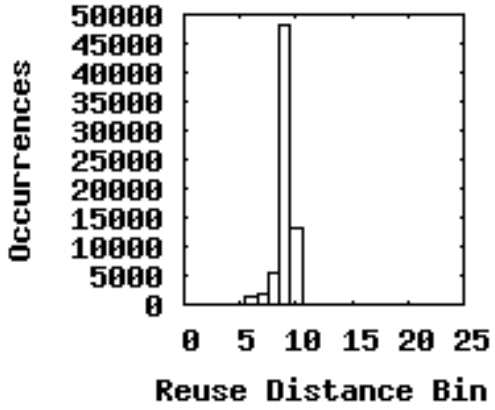

Figure 4.12: PEBS histogram for 445.gobmk instruction responsible for largest number of misses

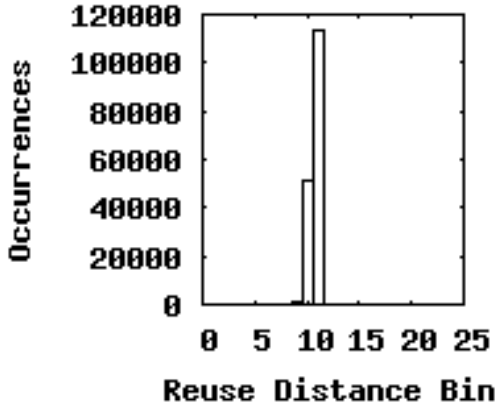

Figure 4.13: PEBS histogram for 447.dealII instruction responsible for largest number of misses

from the `SSVector::setup_and_assign()` function, six from the `SSVector::assign2productFull()` function, and five from `CLUFactor::solveLleftNoNZ()`.

4.1.14 453.povray

There are a few small groups of Pin histograms which follow similar shape, but no single shape is a genuinely common occurrence. Despite this variety, the PEBS histograms show only one common shape, fitting entirely into bins 0 through 4 (see figure 4.14).

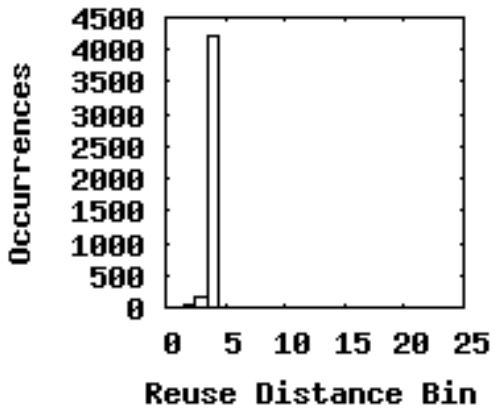

Figure 4.14: 453.povray typical PEBS histogram

4.1.15 454.calculix

2 critical instructions have PEBS histograms showing a single pattern with a gradual climb followed by an abrupt peak, but the other 48 have this climb interrupted, leaving two separated patterns. An example of each type is shown in figure 4.15. The Pin histograms corresponding to the two atypical PEBS histograms show slight similarity in shape (see figure 4.16), though no other instruction matches either shape at all.

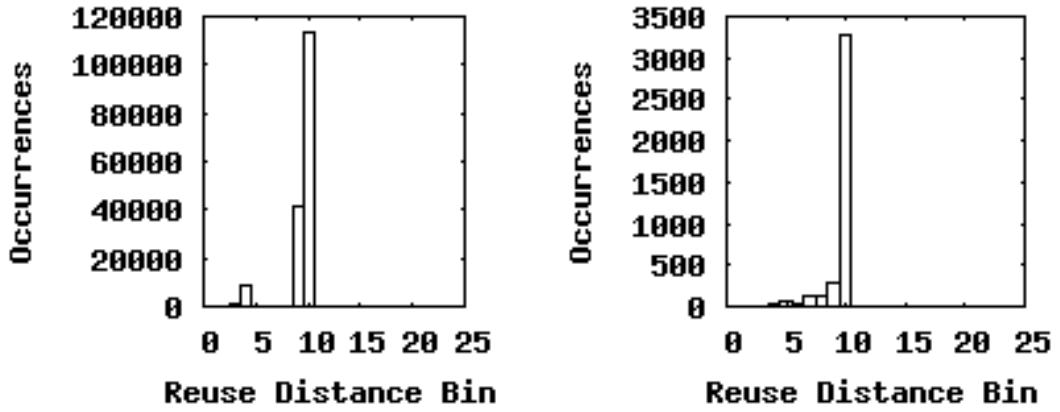

Figure 4.15: 454.calculix PEBS histograms, typical (`0x54fd55` on the left) and atypical (`0x4073db` on the right)

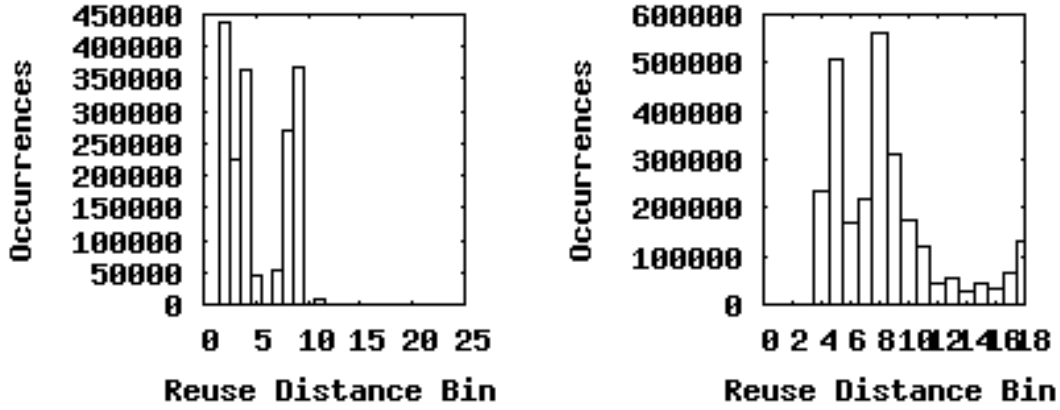

Figure 4.16: 454.calculix Pin histograms corresponding to atypical PEBS histograms (`0x44ff43` and `0x4073db` respectively)

4.1.16 456.hmmer

Pin histograms from this benchmark show some variety, but two shapes occur quite frequently. Both have two disconnected main patterns, with the left-most one gathered in a single bin. They are distinguished by difference in width of the right-most pattern, as seen in figure 4.17. The PEBS histograms all show a single one-pattern shape as in figure 4.18.

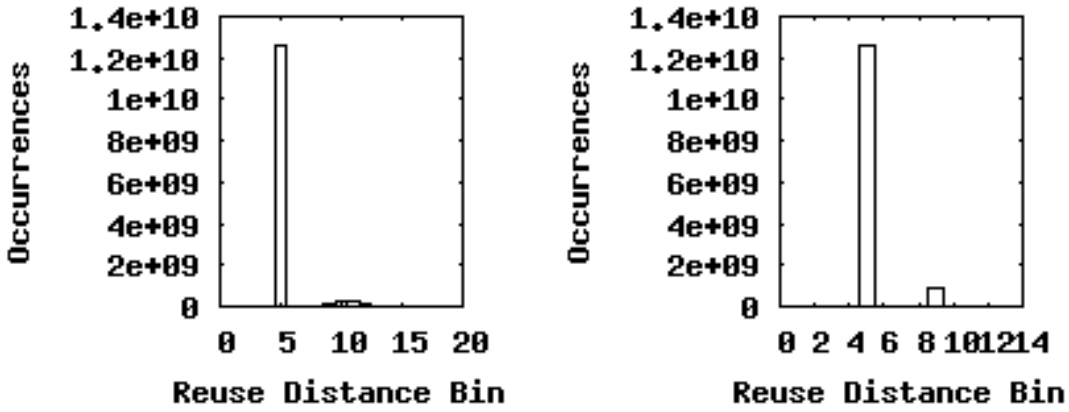

Figure 4.17: 456.hmmer typical Pin histograms

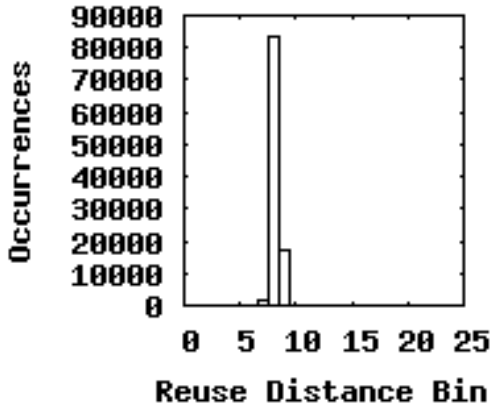

Figure 4.18: 456.hmmer typical Pin histograms

4.1.17 458.sjeng

The first, third, and seventh most frequent miss-causing instructions (totaling 31.8% of the misses) here present two pin histogram patterns, each with gradually sloping sides, so that they intersect with no null between them. An example pin histogram is given in figure 4.19, but the corresponding PEBS histogram has the same shape as those of other critical instructions. One technique used by this chess engine is a hash table mapping previously-

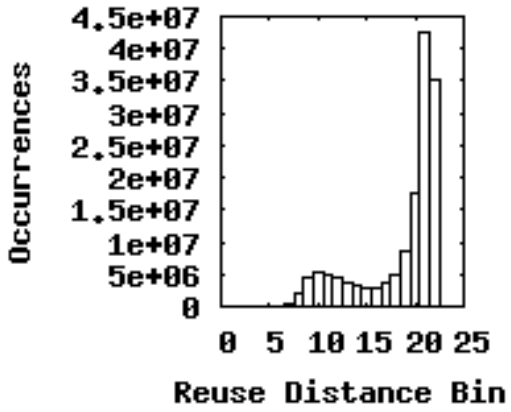

Figure 4.19: 458.sjeng gradual histogram

considered board positions to the moves selected from them. Instructions presenting the described gradual histogram are found in a condition check at the beginning of the function which performs a lookup on this hash table. Although the instructions appear in separate clauses of a short-circuited logical statement, they all exhibit similar histogram shape.

4.1.18 459.GemsFDTD

Again, the PEBS histograms have a single pattern, and the primary difference between histograms is the pattern's left and right taper. The range of this variation is shown in the examples in figure 4.20. This benchmark has several critical instructions with no short-distance reuse (see figure 4.21) as well as common patterns which do have short-distance reuse (given in figure 4.22).

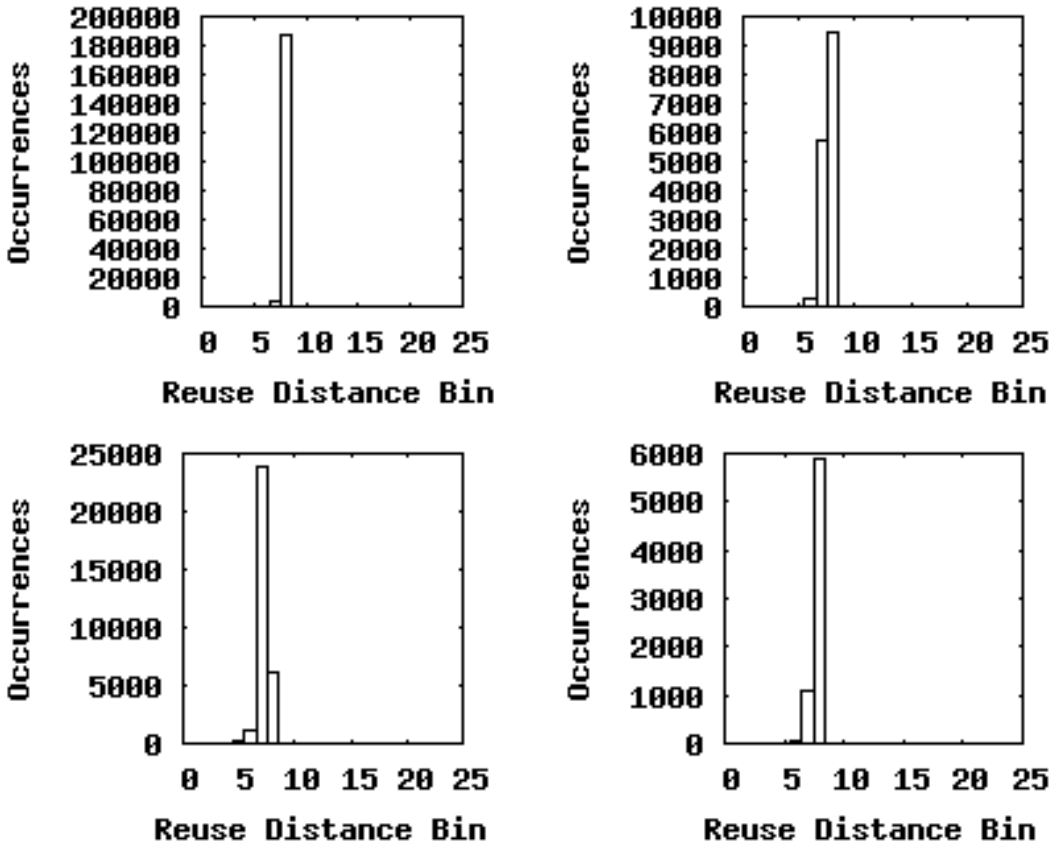

Figure 4.20: 459.GemsFDTD typical PEBS histograms

4.1.19 462.libquamtum

In the case of 462.libquantum, 99.8% of the cache misses were caused by 3 instructions (however, this is a fairly small program, with only 65 instructions appearing in PEBS samples). Each is a reference to the state of a qubit in the quantum register. The quantum register is defined as a `struct` which tracks some general information about itself, e.g. number of qubits it contains, and an array of qubits, each of which is defined by its probability amplitude, a complex float in single-precision, and state, a maximum-length integer

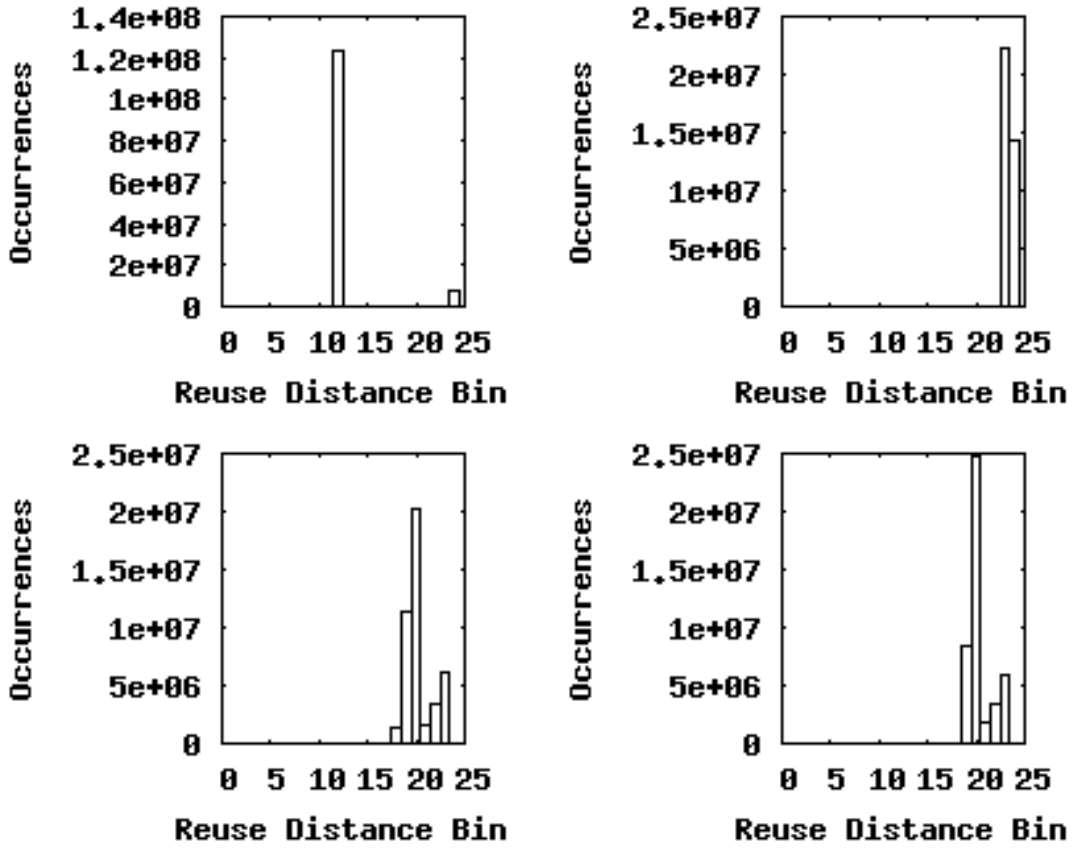

Figure 4.21: 459.GemsFDTD Pin histograms without short reuse distances

type. On the test machine, this totals 16 bytes for the qubit structure, and the `reference` input causes the program to generate a 56-qubit register.

4.1.20 464.h264ref

While several different patterns appear in Pin histograms, many of which appear for multiple critical instructions, only one general shape appears in PEBS histograms. The typical PEBS histogram for this benchmark has an initially slow left-side taper leading up to a

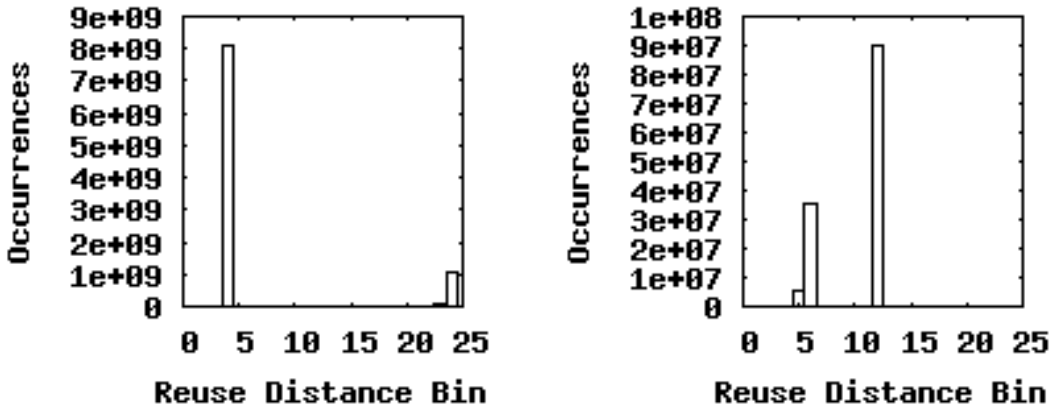

Figure 4.22: 459.GemsFDTD Pin histograms with short reuse distances

peak, usually at the 10th bin, a pattern found in many other benchmarks as well.

4.1.21 465.tonto

This benchmark has a relatively flat distribution of cache misses among its instructions, with its top ten instructions accounting for only 21.9% of the cache misses. While the PEBS histograms for the top ten show the single right-skewed pattern typical of many other benchmarks, wider variation of patterns can be found in PEBS histograms of other instructions. Unfortunately, much of this variation is probably due to statistical noise, as the instructions in the critical set for this benchmark have fewer associated cache misses than those of most other benchmarks: the critical set distributes 62304 miss samples over 424 instructions (averaging 147 misses per instruction). A program like 465.tonto may resist this sort of analysis because too little information is available about individual instructions to have a good prediction of their behavior.

4.1.22 470.lbm

Several of the instructions identified as critical by cache miss sampling have pin histograms with no long-distance reuses. Every critical instruction, except one, comes from the `LBM_performStreamCollide` function, which performs a step of fluid dynamics simulation. The remaining one comes from the `LBM_showGridStatistics` function, which is used to display intermediate results. The reason for this disparity in measured reuse distance (an instruction with only small reuse distances should only appear in PEBS samples for non-reuse occurrences, i.e. compulsory misses) is uncertain.

4.1.23 471.omnetpp

Most critical instructions show the pattern given in figure 4.23, but several present patterns other than this common one (figure 4.24).

4.1.24 473.astar

This benchmark's critical set includes only 13 instructions. 10 of the 13 PEBS histograms show two patterns with a narrow space between them (see figure 4.25). Most variation appears in the tapering of the left pattern, whereas the right pattern is confined to a single

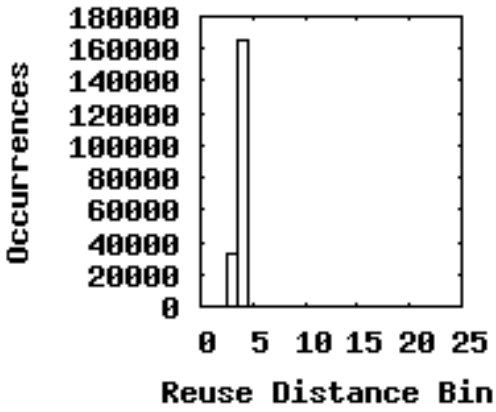

Figure 4.23: 471.omnetpp PEBS histograms with usual shape

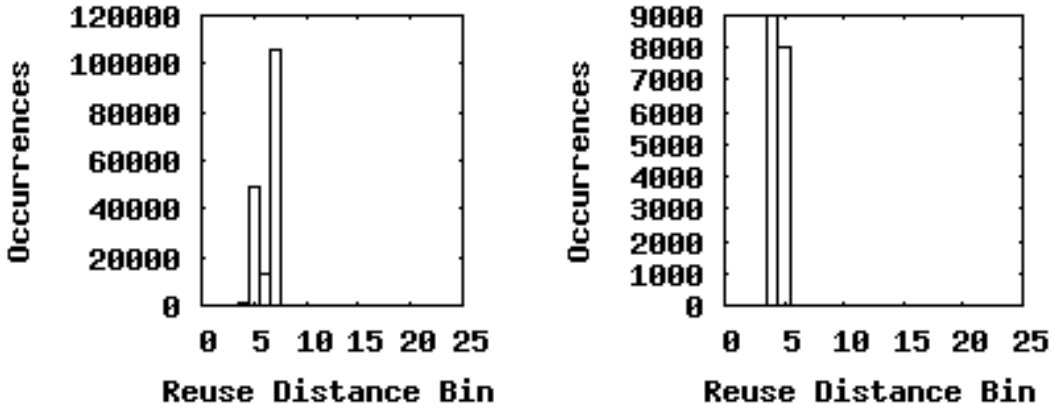

Figure 4.24: 471.omnetpp PEBS histograms with unusual shapes

bin.

4.1.25 481.wrf

Of the 306 critical instructions, all but 16 instructions from 5 functions show the typical shape (figure 4.26), with the peak at the 10th or 11th bin. The remaining 16 have two separated groupings with one or two patterns each (figure 4.27).

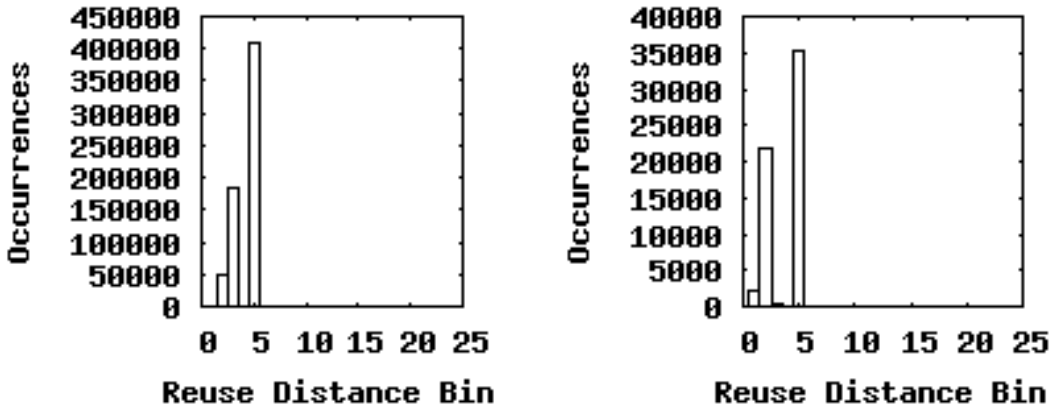

Figure 4.25: 473.astar example PEBS histograms

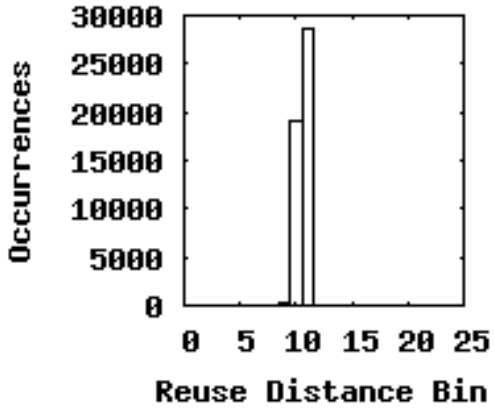

Figure 4.26: 481.wrf typical PEBS histogram

4.1.26 482.sphinx3

A wide range of Pin histogram shapes for a critical set of 31 instructions all correspond to the common single-pattern shape seen in figure 4.28. Two of the recurring Pin shapes are given in figure 4.29. 9 of the critical instructions (accounting for 39.1% of the miss samples) belong to the fairly small (static size of 179 instructions) `mgau_eval` function.

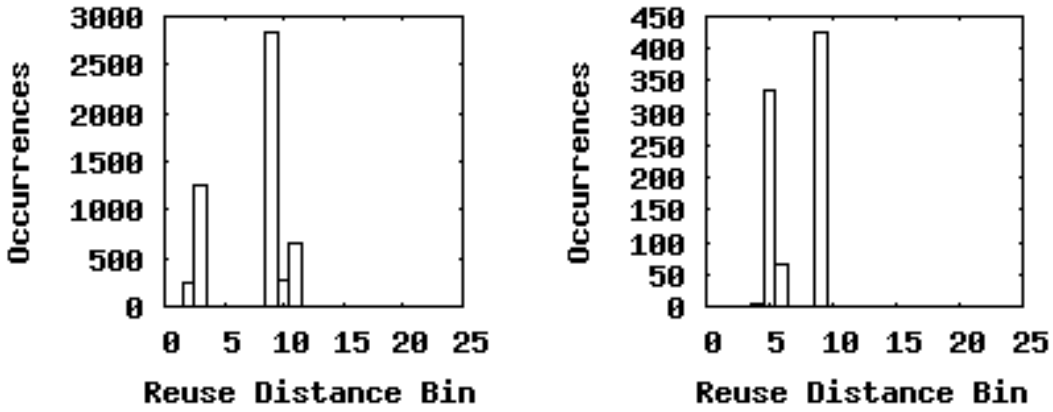

Figure 4.27: 481.wrf PEBS histograms with multiple patterns

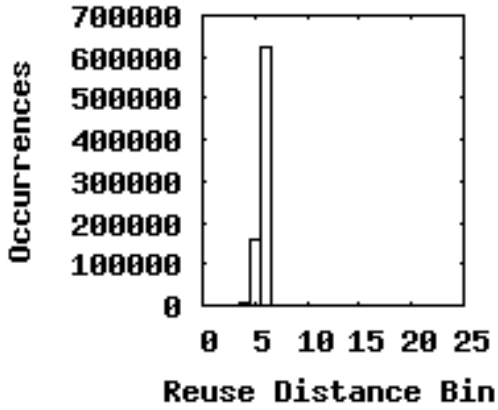

Figure 4.28: 482.sphinx3 typical PEBS histogram

4.1.27 483.xalancbmk

This benchmark has a rather small critical set, 14 instructions, despite having 1284 instructions which generated cache miss samples. All critical instructions but one show the same general shape in their PEBS histograms (see figure 4.30). Two repeated shapes appear in Pin histograms, with examples of each shown in figure 4.31.

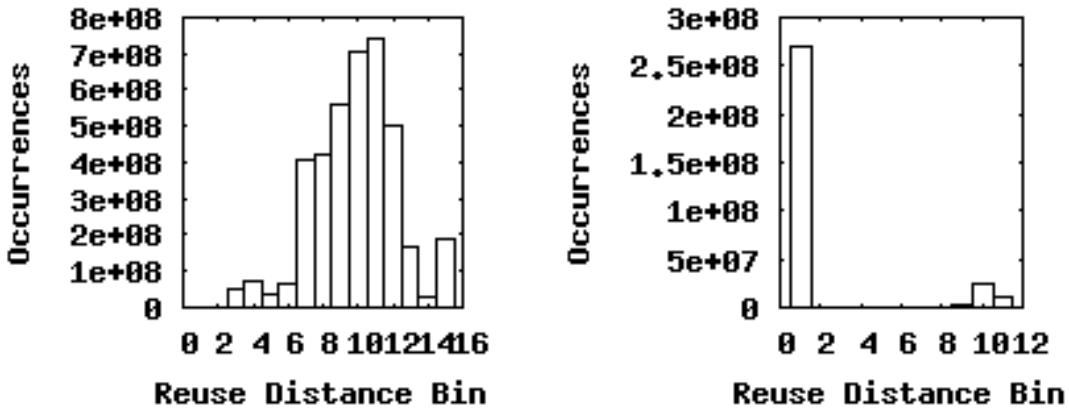

Figure 4.29: 482.sphinx3 typical PEBS histogram

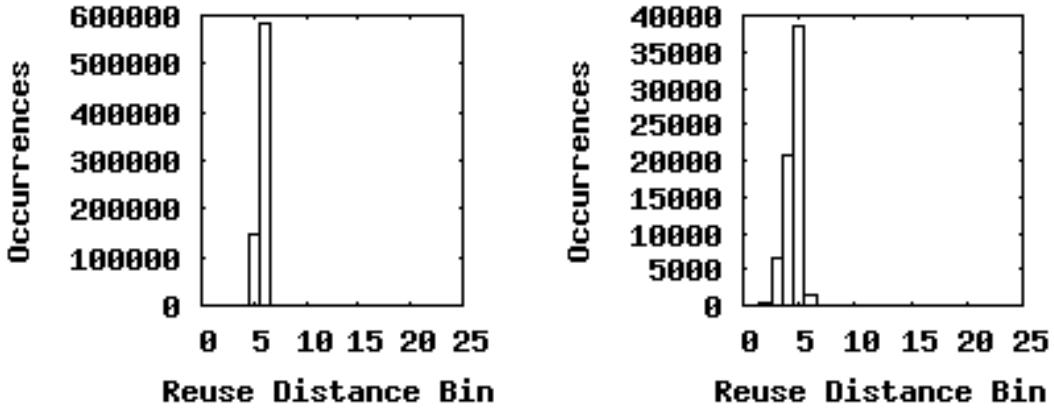

Figure 4.30: 483.xalancbmk typical (left) and atypical (right) PEBS histograms

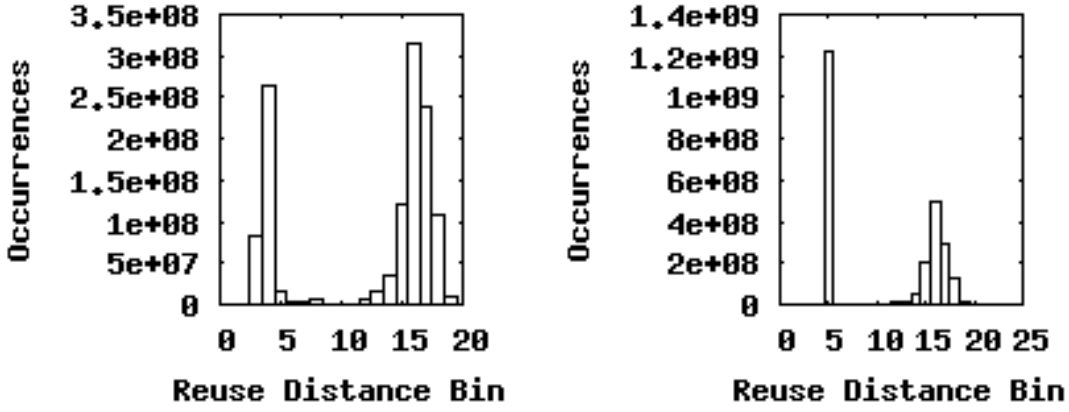

Figure 4.31: 483.xalancbmk typical (left) and atypical (right) PEBS histograms

Chapter 5

Conclusion

5.1 Future Work

While this hardware-based sampling technique makes gathering raw memory access data much quicker, a fast way to construct an instruction and memory address trace is still needed. In this case, the slow step is in identifying the miss-causing instruction when what is given is the instruction immediately after it and an assembly dump of the program. As the PEBS monitor executes as a different process, it does not have the ability to directly read the memory of the monitored program, but some speedup may be achieved by reading from the executable file and disassembling only the code surrounding the instruction identified by the PEBS sample. This entire problem can be avoided on certain CPUs, such as the Itanium, on which a PEBS sample includes the address of the miss-causing instruction and

the memory address it was trying to access (whereas this information must be reconstructed on an Intel Core CPU) [3].

The more difficult of the remaining problems comes from the lack of variation among PEBS histograms: there is no obvious way to link the shape of an instruction's PEBS histogram with the shape of the instruction's Pin histogram. Training runs might be used to establish the Pin histogram's shape, but this has proven problematic in cases where the smaller data size for the training input eliminates long-distance reuse patterns from the histogram. If the PEBS histogram is more strongly linked to the program than to per-instruction memory use patterns, whole-program analysis may still benefit from PEBS-based cache miss analysis. A radically different data collection technique may be necessary in order for hardware performance monitoring to be usable for reuse distance analysis.

A different technique for processing the instruction and memory address trace may prove useful. As in 4.1.10, narrower bins may help distinguish between similarly-shaped histograms. It may also be possible to find a correlation between some aspect of the PEBS histogram and a part of the Pin histogram large enough to still be useful (e.g. the location of the longest-distance pattern can still be useful for prefetching or for estimating a program's miss-rate curve).

5.2 Potential Applictions

Some compiler optimization techniques can already take advantage of reuse distance prediction, but the time needed to construct a reuse distance profile make them prohibitive for use other than static compilation. These include scheduling of memory instructions with the knowledge that certain loads (i.e. those unlikely to cause cache misses) need not be moved as early and insertion of prefetch instructions for loads that are likely to cause cache misses. Constructing reuse data via hardware monitoring rather than instrumentation makes these options available to a JIT compiler, similar to the technique demonstrated by Cuthbertson et al. [4].

While reuse distance prediction can be used to identify critical instructions for arbitrary input, as by Fang et al. [7], this can be subsumed by the process of collecting PEBS data: for a given run of the program, the critical instructions, i.e. those which generate the most cache misses, will be those which generate the most cache miss samples. This is a much simpler prediction to make from PEBS data than the reuse distance patterns of particular instructions. Miss rate prediction, which does require more than identifying the critical instructions, may be useful when attempting to efficiently allocate cache space among multiple running programs, and hardware-based monitoring could make reuse distance analysis fast enough to be a viable strategy for predicting the live programs' miss rate curves.

References

[1] Perfmon2 project. `http://perfmon2.sourceforge.net/`.

[2] B. R. Buck and J. K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[3] B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the intel ltanium 2 processor. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 58–, Washington, DC, USA, 2004. IEEE Computer Society.

[4] J. Cuthbertson, S. Viswanathan, K. Bobrovsky, A. Astapchuk, and E. K. U. Srinivasan. A practical approach to hardware performance monitoring based dynamic optimizations in a production jvm. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 190–199, Washington, DC, USA, 2009. IEEE Computer Society.

[5] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38:245–257, May 2003.

[6] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2004 workshop on Memory system performance*, MSP '04, pages 60–68, New York, NY, USA, 2004. ACM.

[7] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 27–37, Washington, DC, USA, 2005. IEEE Computer Society.

[8] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD'07*, pages 245–250, 2007.

[9] G. Keramidas, P. Petoumenos, and S. Kaxiras. Where replacement algorithms fail: a thorough analysis. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 141–150, New York, NY, USA, 2010. ACM.

[10] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *Ibm Systems Journal*, 9:78–117, 1970.

[11] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 373–382, New York, NY, USA, 2007. ACM.

[12] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 91–100, New York, NY, USA, 2008. ACM.