Decomposing Memory Performance: Data Structures and Phases

Kartik K. Agaram Stephen W. Keckler Calvin Lin Kathryn S. McKinley
Department of Computer Sciences, University of Texas at Austin

Abstract

The memory hierarchy continues to have a substantial effect on application performance. This paper explores the potential of high-level application understanding in improving the performance of modern memory hierarchies, decomposing the often-chaotic address stream of an application into multiple more regular streams. We present two orthogonal methodologies. The first is a system called DTrack that decomposes the dynamic reference stream of a C program by tagging each reference with its global variable or heap call-site name. The second is a technique to determine the correct granularity at which to study the global phase behavior of applications. Applying these twin analysis methods to twelve C SPEC2000 benchmarks, we demonstrate that they reveal data structure interactions that remain obscured with traditional aggregation-based analysis methods. Such a characterization creates a rich profile of an application's memory behavior that highlights the most memory-intensive data structures and program phases, and we illustrate how this profile can lead system and application designers to a deeper understanding of the applications they study.

Categories and Subject Descriptors B.8 [*Performance and Reliability*]: Performance Analysis and Design Aids; C.4 [*Performance of Systems*]: Measurement techniques

General Terms Design, Experimentation, Measurement

Keywords Simulation, Data structure, Phase, SPEC, CPU2000, DTrack

1. Introduction

As a result of application and computer system design trends, the memory system continues to exert a dominant influence on program performance. The importance of memory system behavior will continue to grow as the gap between memory speeds and processor speeds increases. In addition, applications are continuing to grow in complexity, which places additional burden on the memory system due to large or irregularly accessed data structures. Understanding how applications use the memory system is important to at least three groups: (1) system designers who can apply insights into memory system usage to improve hardware and software memory optimization techniques, (2) application writers who can understand how their program uses the memory system and optimize for better locality, and (3) benchmark developers who want to ensure that the diverse patterns of behavior in realistic applications are represented. In this study we explore the benefits of program understanding along two orthogonal dimensions - data structure and program phase - and show how such insights can be combined to yield a rich picture of application behavior.

Analyzing memory behavior is a well-trodden field; this paper makes three novel contributions to it. First, we develop a tool called DTrack to decompose the performance of the memory hierarchy by high-level data structure for C programs. DTrack uses a C-to-C compiler to instrument variable allocations, thereby allowing each memory reference to be mapped to a specific global variable or heap call-site. Second, we fill a gap in recent studies on phase behavior: selecting the correct profiling interval for an application, the granularity at which behavior statistics are aggregated. While studies on phase behavior so far use a single, arbitrarily-chosen profiling interval for all applications, we show that the profiling interval is best selected on an application-specific basis. Finally, we apply our methodologies to twelve of the fifteen C benchmarks in the SPEC2000 benchmark suite, and present a detailed characterization of their memory behavior. Our results highlight the wide variety of behaviors exhibited by applications in the distribution of misses by data structure, as well as in the number and interleaving of different phase regimes. Several case studies demonstrate the usefulness of these results in helping the computer architect make sophisticated design decisions.

The remainder of this paper is organized as follows. Section 2 distinguishes our work from prior memory system analysis studies and tools. Section 3 describes DTrack and demonstrates its ability to decompose the aggregate behavior of applications by data structure. This Section also describes two case studies that illustrate the uses of DTrack in framing and rapidly answering sophisticated questions in designing new systems. Section 4 takes our analysis further, decomposing the address streams of applications by both data structure and time. A major contribution here is a new way to determine the right granularity at which to sample phase data, and we show that this granularity varies from application to application. Finally, Section 5 provides conclusions and thoughts on future work.

2. Related work

Conventional methodology for characterizing applications involves either cache or timing simulation [1, 9, 22, 3]. These techniques operate at the level of the application executable without recourse to the high-level structure of the program. As a result, their output is limited to aggregate statistics about hardware execution, such as the mean number of instructions executed per clock cycle, miss-rates at the various cache hierarchies, and similar *hardware* events. In this paper we decompose these aggregate statistics by data structure and by program phase. We now review the prior work in each of these areas.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-221-6/06/0006. . . $5.00.

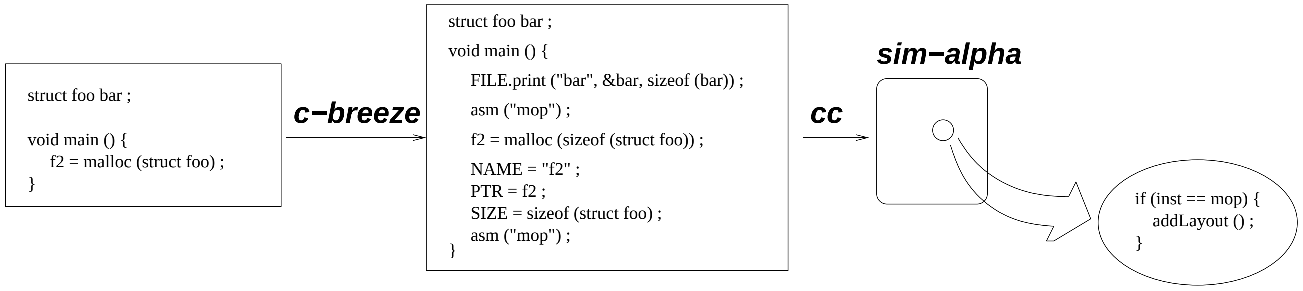

Figure 1. DTrack, a tool for observing data structures in programs

2.1 Decomposing memory behavior by data structure

Tools have been created in the past to decompose application memory performance by data structure, but they have thus far been restricted to studying arrays. The original such tool is MemSpy by Martonosi et al. [13] which operated on loop nests in Fortran programs. Similarly, Lebeck et al. [12] present data structure and procedure level aggregate miss information and classify misses as compulsory, capacity and conflict. While these tools present several software optimizations for improving cache performance, they examine the behavior of an array within the context of a single procedure. As a result, they do not perform cross data structure analysis. For example, they do not consider the question of whether data structures interfere with themselves or with others.

McKinley and Temam analyze the complementary dimension of inter-nest and intra-nest loop locality [14], but again consider only arrays and aggregate information between loop nests. Seidl and Zorn [18] use a technique similar to ours to partition the heap into segments based on object lifetimes, without performing the more fine-grained analysis to separate the behavior of objects by data structure that we do.

2.2 Analyzing time-varying behavior and detecting phases

Several tools have studied time-varying behavior. The Cache Visualization Tool [25] demonstrates the time-varying behavior of arrays as they march through the cache. This level of detail supports analyzing a single loop nest at a time, whereas we analyze data structure phase behavior across much longer periods. Chilimbi et al. [4, 17] analyze compressed program traces, decompose them into *hot data streams*, and use these hot data streams to drive layout and prefetching optimizations. This approach of searching for access patterns across the different data structures in a program is complementary to ours, which attempts to decompose application access patterns by data structure. We believe our approach is more effective at providing intuitions about application behavior that are useful to humans in different roles.

More recently, several studies have used some form of code signature to detect phase boundaries. Basic Block Vectors (BBVs) are currently the most accurate method to generate code signatures, and several studies explore their uses in clustering phases and detecting phase transitions in an offline [20, 19] and online [21] setting. One alternative to BBVs is the use of program counter or Extended Instruction Pointer Vectors (EIPVs) [2], whose merits have been debated by Lau et al. [10]. Another alternative consists of more high-level metrics based on code structure, such as register use vectors or loop vectors [11]. All these studies, however, select an arbitrary sampling period and use it for all the applications they evaluate. In this study we provide a more rigorous method to separately determine the correct sampling period for each application.

Perhaps the most similar work to ours is the online phase detector of Nagpurkar et al. [15]. Their system maintains a current window of object references within a JVM and assesses the similarity of the recent references in it to those in an older trailing window. Like our study they evaluate the effect of window size (sampling interval) on phase detection. While our study looks for phases in fine-grained behavioral statistics of an application, they study phase behavior in the functional list of object references touched by an application. The two approaches are complementary.

3. Decomposing behavior by data structure

This section describes DTrack and our methodology for analyzing applications, and performs a detailed analysis of the data structures of twelve applications. DTrack maps addresses to data structures by automatically inserting instrumentation in the application to communicate the address range corresponding to each variable to the simulator. The challenge here is to keep the overhead due to the instrumentation low and to minimize the perturbance to the application. Figure 1 shows a schematic of our tool. First, we automatically instrument benchmark sources using an extension to the C-Breeze [7] C-to-C compiler. We then simulate them on a modified version of the sim-alpha [6] timing simulator that simulates the configuration shown in Figure 2, including a Rambus memory model. For each variable in the program, the compiler-generated instrumentation stores the variable's name and address at a designated location in memory and interrupts the simulator by means of a special opcode ("mop" in Figure 1). On executing this instruction at runtime, the simulator imports the information from this designated location in simulated memory. Since the simulator knows the extent of each variable in the application at any time, it maps the address of each cache access to a specific variable. Classifying and assigning each load and store to a specific variable slows the simulator down by 60% on average and 100% in the worst case.

We track both heap allocations and deallocations because the same raw address could be allocated to different data structures at different times in a program's execution. Since we classify heap allocations according to their static location in the source code, we cannot distinguish between instances of a data structure, such as two linked lists whose nodes are allocated at the same line in the source. This issue is not a concern in studying the C SPEC2000 benchmarks because the major data structures do not have multiple instances. Other languages and benchmarks may require more elaborate heuristics. Global variables are handled differently. Rather than communicate them individually to the simulator by the above method, the instrumentation writes the names and extents of all global variables to a designated file on program initialization. Though the set of file writes is expensive, it is a one-time startup cost. Finally, stack variables are not instrumented because the high frequency of scope changes would raise the instrumentation overhead too much. Instead, we treat the stack as a single data structure and coalesce all accesses to it by a simple range test. Our results below show that misses to the stack are generally negligible. We

Feature	Size/Value
Data caches	
DL1 cache	64 KB, blocksize 64 bytes, 2-way, 3 cycles
L2 cache	512 KB, blocksize 64 bytes, direct-mapped, 12 cycles
TLBs	128 entries
Main memory	
Peak bandwidth	1.6Gbytes/s
Rambus geometry	64 banks * 512 rows * 2KB/row
Access latency (cycles)	32 PRER + 24 ACT + 48 RD/WR + queuing
Out-of-order Processor	
Pipeline width	4
Int ALUs, multipliers	4,4
FP ALUs, multipliers	1,1
Branch predictor	Tournament, 1 KB x 1 KB local, 4 KB global, 4 KB choice

Figure 2. Details of the simulated Alpha 21264-like processor and memory hierarchy

Benchmark	IPC	DL1 Miss-rate	L2 Miss-rate
164.gzip	1.39	2.3	3.9
175.vpr	0.67	3.0	35.3
176.gcc	1.15	3.2	10.4
177.mesa	1.06	0.9	23.4
179.art	0.23	14.8	74.9
181.mcf	0.14	24.1	60.5
183.equake	0.58	14.1	29.4
186.crafty	1.21	1.3	4.3
188.ammp	0.57	10.0	45.0
197.parser	0.97	3.6	21.5
256.bzip2	1.16	2.1	32.6
300.twolf	0.51	9.5	26.9

Figure 3. The benchmarks we use and their aggregate memory hierarchy behavior

verify that our instrumentation does not perturb application behavior; dynamic instruction counts increase by less than 0.6% across all benchmarks except for 164.gzip, where the instrumentation is 3.7% of the total instruction count because of frequent heap allocations in the inner loops.

3.1 Benchmarks, inputs and simulation periods

We now describe our methodology for performing detailed analyses of applications from a memory system perspective. First we describe techniques to map addresses to data structures while minimizing the degree to which we perturb underlying application behavior. We then move to phase analysis and describe our technique for selecting the profile period for each of our applications. Finally, we describe the machine configuration we simulate in our characterization, the simulation periods we choose, and the aggregate statistics for our benchmarks that can be gleaned from conventional tools.

This paper presents a characterization of twelve of the fifteen C benchmarks in the SPEC2000 benchmark suite. Figure 3 lists some aggregate properties of the benchmarks we study, including average instructions per cycle (IPC) and miss-rates at the level-1 data (DL1) and level-2 (L2) caches. Our benchmarks range from regular ones such as 179.art to highly irregular ones such as 300.twolf, from compute-bound (164.gzip) to memory-bound (181.mcf). We are unable to study the remaining 3 C benchmarks in the SPEC2000 suite due to methodological difficulties; 253.perlbmk no longer builds on our Alpha platform with the latest version of libc, and 254.gap and 255.vortex run incorrectly on our native Alpha platform because of unaligned addresses generated by their custom memory-managers. While these unaligned addresses could be fixed by modifying the benchmark sources, we estimate that adding the necessary padding could significantly perturb benchmark behavior.

All our simulations use the designated ref input set for the corresponding benchmark. We demarcate the end of initialization by a special opcode using the techniques outlined above, and perform fast functional simulation until we reach this opcode. Thereafter we perform detailed timing simulation for 500 million instructions. These simulation periods are representative of each application's runtime, as determined in the course of our study of global phase behavior later in this paper.

3.2 Data profiles and distributions

Having described DTrack and our experimental methodology, we now present a detailed characterization of the above SPEC benchmarks using DTrack. We begin by studying basic data profiles generated by DTrack, and then explore two ways that this new capability to visualize the behavior of different data structures can be used to help answer sophisticated architectural questions.

DTrack generates data profiles. Figure 4 breaks down the aggregate memory behavior of our applications – accesses and miss-rates at the DL1 and L2 – by the three data structures that cause the most DL1 misses (DS1, DS2, DS3), the stack, and everything else. Figure 4.a shows that the breakdown of accesses to the DL1 (and therefore the rest of the memory hierarchy) varies greatly across our applications. While 179.art and 181.mcf have skewed distributions, with 80% of all accesses coming from 2 data structures, 176.gcc and 186.crafty have extremely balanced distributions; no data structure contributes more than 2% of accesses. Other applications lie between these extremes.

While accesses are often spread out, Figure 4.b shows that misses tend to cluster. The top 5 data structures usually contribute more than 90% of all DL1 misses. The exceptions are 176.gcc, 186.crafty, and 197.parser with a long tail of minor data structures that respectively end up accounting for 84%, 67% and 78% of all cache misses. Among the other applications, the major data structures end up partitioning cache misses among themselves in a variety of ways; the top data structure can contribute anywhere between 20 and 80% of total cache misses.

Comparing Figures 4.a and 4.b, we see that cache misses and accesses are poorly correlated. A few applications such as 179.art and 181.mcf reveal a simple underlying organization with only a few data structures, and misses tracking the distribution of accesses. However, the majority of applications show a well-understood pattern where a data structure receives more accesses than another, yet accounts for fewer misses. In particular, the stack accounts for a significant fraction of accesses without ever presenting a significant problem to the DL1. The sole exception is 186.crafty where the stack collectively contributes more misses than any single global data structure. As we have seen, however, 186.crafty has a very balanced distribution, and the stack still accounts for only 11% of DL1 misses.

3.3 Access pattern variety

So far we have looked at differences in miss distribution across the major data structures in the different SPEC benchmarks while hiding details about the individual data structures behind the anonymous names DS1, DS2 and DS3. Figure 5 now summarizes the

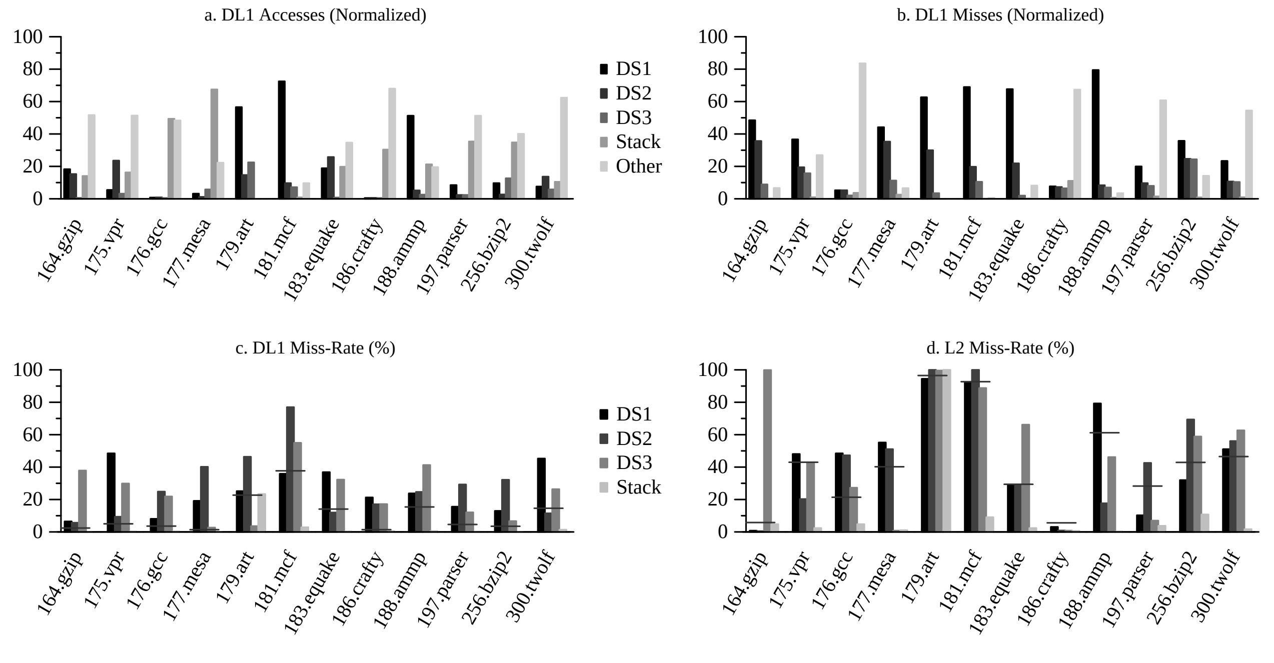

Figure 4. Decomposition of DL1 and L2 behavior by data structure. Horizontal lines in the miss-rate graphs indicate the aggregate miss rate for each benchmark across all data structures. L2 misses show similar trends to DL1 misses.

high-level details of these data structures. For each benchmark, we show the name of these data structures as used in the source code, along with a brief summary of the type of the data structure (array or recursive), whether it is predominantly accessed in a *regular* fashion with spatial locality or in an irregular fashion with low spatial locality. Finally, we provide the size of each object in these data structures and their total sizes in the application.

Figure 5 shows that the major data structures are predominantly array-based in the applications we study. However, these data structures are often used to emulate complex graphs using either real pointers (181.mcf:`nodes`, 175.vpr:`rr_node`) or integers that index into other arrays (256.bzip2:`quadrant`, 300.twolf:`rows`). The wide variety of uses indicate that data structures are often declared to be arrays solely to simplify memory management. Most of the major data structures are dynamically allocated on the heap. The major exceptions are 186.crafty that causes a significant fraction of misses to the global segment, and 176.gcc which allocates most of its variables on the stack using `alloca`.

While 179.art and 183.equake have regular access patterns, the others interleave spatial and pointer access in complex ways. This interleaving may happen either because of strided access through an array while dereferencing pointer fields from each element (mcf:`nodes`, 188.ammp:`atoms`), or because of strided access that uses the elements of one array to index into another (bzip2:`quadrant`, 300.twolf:`rows`) in a form of pointer traversal that current pointer prefetching schemes [16, 5] often cannot detect, or finally because the program accesses the elements of a data structure in irregular order, but each object spans multiple cache blocks that are accessed sequentially (ammp:`nodelist`, twolf:`netptr`) due to large object size or irregular object alignment in the cache. Such complex interleavings are a challenge to both spatial and pointer-based prefetch systems.

Having used the basic capabilities of DTrack to characterize our applications, we now explore novel uses of DTrack in asking and answering sophisticated questions on architecture design.

3.4 Case study: Data structure criticality

Our first case study concerns criticality of memory reference. Several recent studies have shown that not all cache misses are equally important as measured in the amount of latency that they expose to the processor [24]. In this context, does it make sense to simply use miss counts to select the data structures on which to focus our attentions? To answer this question we augment DTrack to detect cycles when no instructions are retired, and assign responsibility for each such *stall cycle* to the data structure referenced by the load or store at the head of the reorder buffer [23]. Our results show that for our applications the data structures that cause the most misses are almost always also the ones responsible for the most stall cycles. There are two exceptions to this trend. The first is in 179.art; the array `tds` causes only 2.1% of all cache misses, but is responsible for 16.6% of all stall cycles. This data structure is critical because of the following loop that accumulates a subset of its elements:

```
for (tj=0;tj<numf2s;tj++) {
    if ((tj == winner)&&(Y[tj].y > 0))
        tsum += tds[ti][tj] * d;
}
```

This combination of data-dependent branches and computation serialized by `tsum` causes the infrequent cache misses in this loop to almost invariably stall the pipeline. Our conclusion is strengthened by a study of the source code. 179.art is a neural network simulator where learning occurs by iteratively modifying two arrays of top-down and bottom-up weights – `tds` and `bus` respectively. While these two arrays are largely accessed in very similar ways, the above loop is the only major access pattern not shared with `bus`. The second data structure that we observe causing a disproportionate number of stalls is the variable `search` in the chess-playing benchmark 186.crafty, which is responsible for 10.5% of all stall cycles in spite of causing just 0.2% of all cache misses. This global data structure contains the chess position being currently analyzed, and is used to display the board on screen. With the exception of

Benchmark	DS1	DS2	DS3
164.gzip	`window` *array – regular* 64 KB in 2-byte objects	`prev` *array – regular* 64 KB in 2-byte objects	`fd` *array – regular* 184320 KB in 1-byte objects
175.vpr	`rr_node` *array – irregular* 10638 KB in 40-byte objects	`heap` *array – irregular* 6717 KB in 24-byte objects	`rr_node_route_inf` *array – irregular* 2653 KB in 16-byte objects
176.gcc	`reg_last_sets` *array – irregular* 0.5 KB in 8-byte objects	`reg_last_uses` *array – irregular* 0.5 KB in 8-byte objects	`qty_const_insn` *array – irregular* 4 KB in 8-byte objects
177.mesa	`Image Buffer` *array – regular* 2560 KB in 2-byte objects	`Depth Buffer` *array – regular* 5120 KB in 4-byte objects	`Vertex Buffer` *array – regular* 920 KB in 1 object
179.art	`f1_layer` *array – regular* 625 KB in 64-byte objects	`bus` *array – regular* 859 KB in 8-byte objects	`tds` *array – regular* 859 KB in 8-byte objects
181.mcf	`nodes` *array – regular & irregular* 7071 KB in 120-byte objects	`arcs` *array – irregular* 188416 KB in 64-byte objects	`dummy_arcs` *array – irregular* 3771 KB in 64-byte objects
183.equake	`K` *3D array – regular* 22399 KB in 8-byte objects	`disp` *3D array – regular* 2828 KB in 8-byte objects	`M` *2D array – regular* 943 KB in 8-byte objects
186.crafty	`rook_attacks_rl90` *array – irregular* 128 KB in 8-byte objects	`last_ones` *array – irregular* 64 KB in 1-byte objects	`first_ones` *array – irregular* 64 KB in 1-byte objects
188.ammp	`atoms` *pointer – regular & irregular* 41322 KB in 2208-byte objects	`nodelist` *array – regular* 76 KB in 232-byte objects	`atomlist` *array – regular* 4372 KB in 232-byte objects
197.parser	`Connector` *various – irregular* variable allocation in 24-byte objects	`Disjunct` *various – irregular* variable allocation in 40-byte objects	`table` *various – irregular* variable allocation in 40-byte objects
255.bzip2	`block` *array – irregular* 900 KB in 1-byte objects	`quadrant` *array – irregular* 1800 KB in 2-byte objects	`zptr` *array – irregular* 3600 KB in 4-byte objects
300.twolf	`net_array[]`$\rightarrow$`netptr` *pointer – irregular* 253 KB in 48-byte objects	`tmp_rows` *array – irregular* 34 KB in 1-byte objects	`rows` *array – irregular* 34 KB in 1-byte objects

Figure 5. Descriptions of the major data structures in Figure 4. Information on each benchmark for each major data structure: container type, access pattern, container and element size.

these two data structures, the correlation between miss count and stall cycle count shows that data-structure criticality is of limited usefulness in the predominantly irregular programs that we study.

A related idealization experiment that provides indirect confirmation of this result explores the effect of selectively providing different data structures perfect single-cycle access to memory. To model this ideal behavior we simulate cache misses to specific data structures in a single cycle, but continue to move data in these structures through the memory hierarchy so as to not give other data structures an unrealistically generous view of cache capacity. We find that selectively eliminating cache misses in even the most important data structure in an application has limited impact on bottomline performance in a majority of our applications. While there are a few exceptions, namely 188.ammp, 183.equake, it usually requires perfect memory for 2-5 major data structures to bring performance close to ideal. This result shows that future architectural and compiler enhancements will often need to optimize multiple data structures in different ways to significantly improve overall performance in memory-bound applications. It also shows that DTrack is indeed highlighting bottlenecks in the memory system when it ranks data structures by miss frequency.

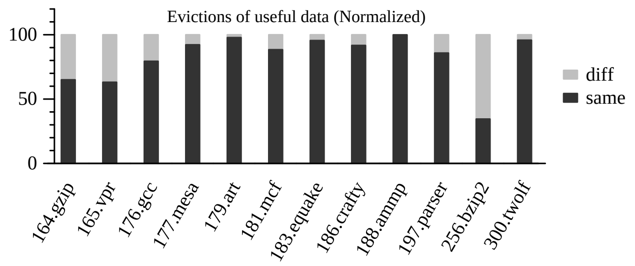

Figure 6. Breakdown of premature evictions. Useful data is only infrequently evicted by a different (diff) data structure.

3.5 Case study: Competition for caches

Where Figures 4.a and 4.b show the distribution of accesses to the DL1 and L2, Figures 4.c and 4.d show the corresponding miss-rates at each level of the memory hierarchy. A common pattern in these figures is for a data structure with fewer cache misses to have a higher miss-rate. This pattern occurs as the major data structures

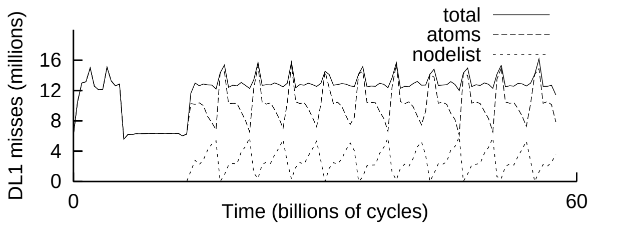

Figure 7. Just tracking total misses can miss interesting effects. DL1 cache misses in aggregate and by data structure in `188.ammp`

compete with each other for limited cache capacity, so that a data structure that misses more often ends up with a larger fraction of the cache. While this is qualitatively a desirable response, such competition may cause suboptimal performance if different data structures repeatedly evict each other. If this behavior were found to be common, a computer architect may consider creating split caches [8] with static mapping policies assigning each data structure to a specific cache partition. Figure 6 shows how often useful data in the cache is prematurely evicted by a different data structure as opposed to the same one. With the exception of `256.bzip2` the majority of premature evictions are caused by conflict within a data structure, rendering a split cache by data structure unnecessary for these applications. This and the previous experiment are good examples of the ways that DTrack can help the computer architect with design decisions where traditional tools are unable to do so.

4. Analyzing data structure phase behavior

The previous Section demonstrated that aggregate statistics of memory performance can hide new interactions between data structures. We now study the global time-varying behavior of these statistics for nine of our twelve applications. Studying phase behavior by data structure is important; looking at the time-varying behavior of aggregate misses alone can be misleading and hide important data structure interactions. Figure 7 illustrates this: the data structures `atoms` and `nodelist` in `188.ammp` are consistently anti-correlated. As one increases the other decreases and vice versa. Studying just the curve for total cache misses would miss this interaction and also underestimate the degree to which the application's behavior is changing under the surface. This pattern is not uncommon; six of our nine applications exhibit significant differences in data structure miss distribution in different phases.

Our analysis demonstrates two broad properties of the phase behavior in our applications. First, observable phase behavior is dependent on the granularity of our observations, so that global trends are often most salient at a very narrow window of *sampling periods* particular to an application. A bad choice of sampling period can underemphasize important global phase transitions or hide them entirely. Second, the phase behavior in most of our applications, when observed at their optimal sampling periods, has a regular structure consisting either of dramatic phase transitions or gradual trends with a well-defined period. We now define necessary terminology, then describe our novel technique for selecting good sampling periods, and finally describe these observations in more detail.

4.1 Process and terminology

Studying phase behavior is an exercise in abstraction. We want to mask out irregularities in the data and focus on the underlying regularities. However, no single sampling period can highlight all the regularities in the data. As Figures 8a,d,g show, the sampling period profoundly affects the nature of observable phase behavior in the data. In this study we focus on global phase behavior, choosing

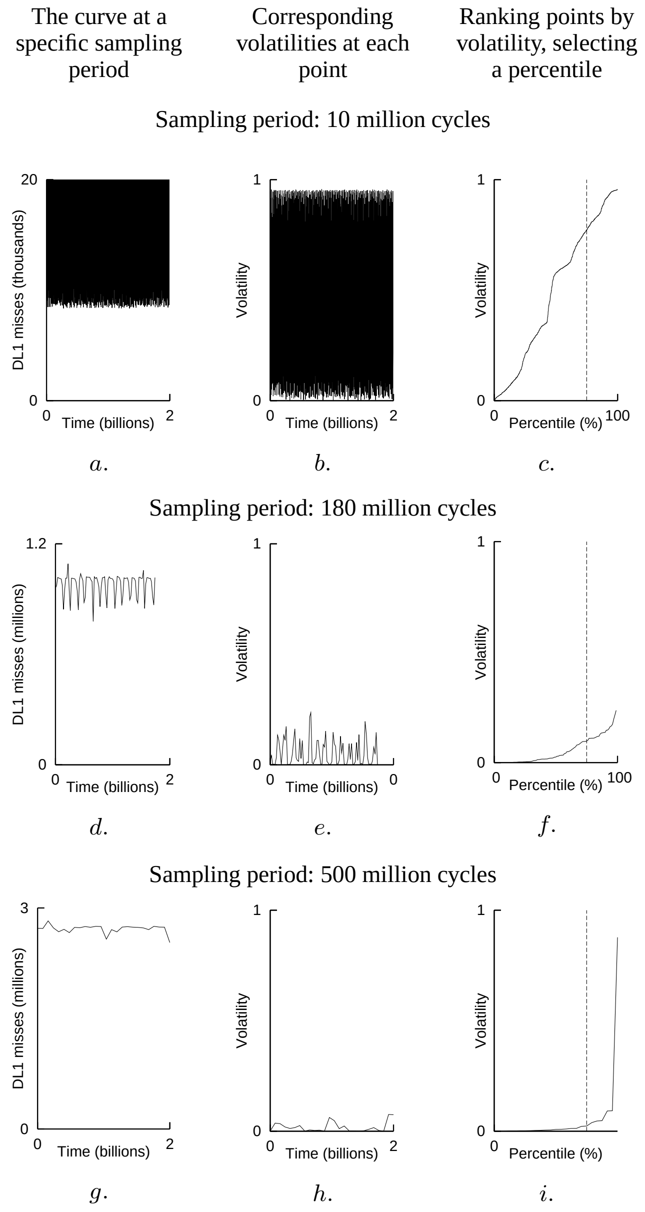

Figure 8. The curves corresponding to a stream at different sampling periods, and their volatilities. (`183.equake`)

to damp out details of fine-grained phase behavior without damping global transitions.

Our process for studying phase behavior is as follows: We modify DTrack to emit time-varying *streams* of miss counts and rates by data structure at a base *sampling period*. In this study all our streams have sampling periods of 1 million cycles. Finer granularities than that generate impractical quantities of data for the large simulation periods we simulate, and we postpone a study of fine-grained local phase behavior at different points in program execution. The streams we generate can be aggregated to simulate *curves* of different sampling periods, generating curves like Figures 8a,d,g. At a specific sampling period, we define and compute a measure of *volatility* at each point on the curve, thus transforming the curve to a corresponding volatility curve as shown in Figures 8b,e,h. We now define the volatility of the curve in terms of these point volatilities by ranking all the points and selecting the

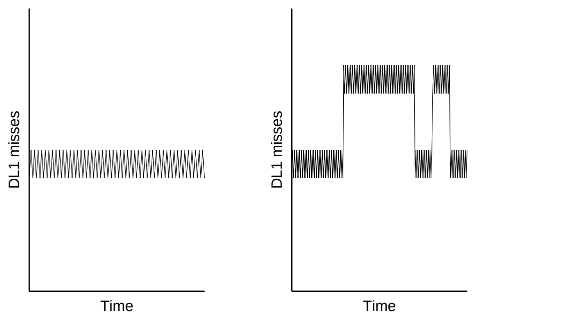

Figure 9. Two example curves that have the same volatility. Our volatility metric is oblivious to coarse-grained phase changes.

volatility of the point at a specific percentile. This process is illustrated in Figures 8c,f,i. Plotting the volatility of curves corresponding to the same stream at different sampling periods yields the *volatility profile* of the stream, and we show how to use the volatility profile to select a sampling period that damps out fine-grained 'noise' but not coarse-grained phase transitions. We are then able to most clearly observe the global phase behavior of our applications.

4.2 Quantifying the volatility of a stream

Volatility is intuively a measure of the change in magnitude of sampled data at adjacent points. We formalize this notion into the following volatility metric at a given time step. Given a stream $X_1, X_2, X_3 \ldots$, the volatility at each time step is defined as:

$$g_t = \frac{abs(X_t - X_{t-1})}{max(X_t, X_{t-1})} \qquad (1)$$

g_t is similar to the conventional notion of 'growth', except that it is symmetric: g_t is 0.5 whether X_t has doubled ("grown by 100%") or halved ("shrunk by 50%") since the last time step. This symmetry ensures that the volatility between two values is the same regardless of whether the curve grows or shrinks between them. Computing g_t at each time step of a curve, we can transform it to yield a curve showing the volatity at each point, as shown in Figures 8a and b.

We now compute the volatility of a curve by ranking the volatility of its points and selecting the volatility at a specific percentile (Figure 8c). As Figure 9 illustrates, this metric has the useful property that it is affected by the volatity of high-frequency 'noise' without taking low-frequency phase transitions into consideration. What constitutes a low-frequency phase transition is dependent on the percentile we use, and we empirically find the 90th percentile to be a reasonable boundary to distinguish between our intuitive notions of phase boundary and noise. That a curve has volatility V thus means that 90% of the points on the curve have volatilities of V or less.

4.3 Volatility profiles and selecting a good sampling period

Given the above volatility metric, we can now study the phase behavior of our applications, and also how this phase behavior varies with sampling period. We summarize the effects of sampling period on phase changes at all granularities by generating a *volatility profile* for each application. The volatility profile for a stream plots the volatility of the curves generated from it at various sampling periods. Across the applications we study, we find that the DL1 and L2 miss counts for different data structures largely exhibit volatility profiles with the same trends, and with minima at the same sampling periods. Figure 10 therefore shows the volatility profile for the DL1 miss stream of a single major data structure in our ap-

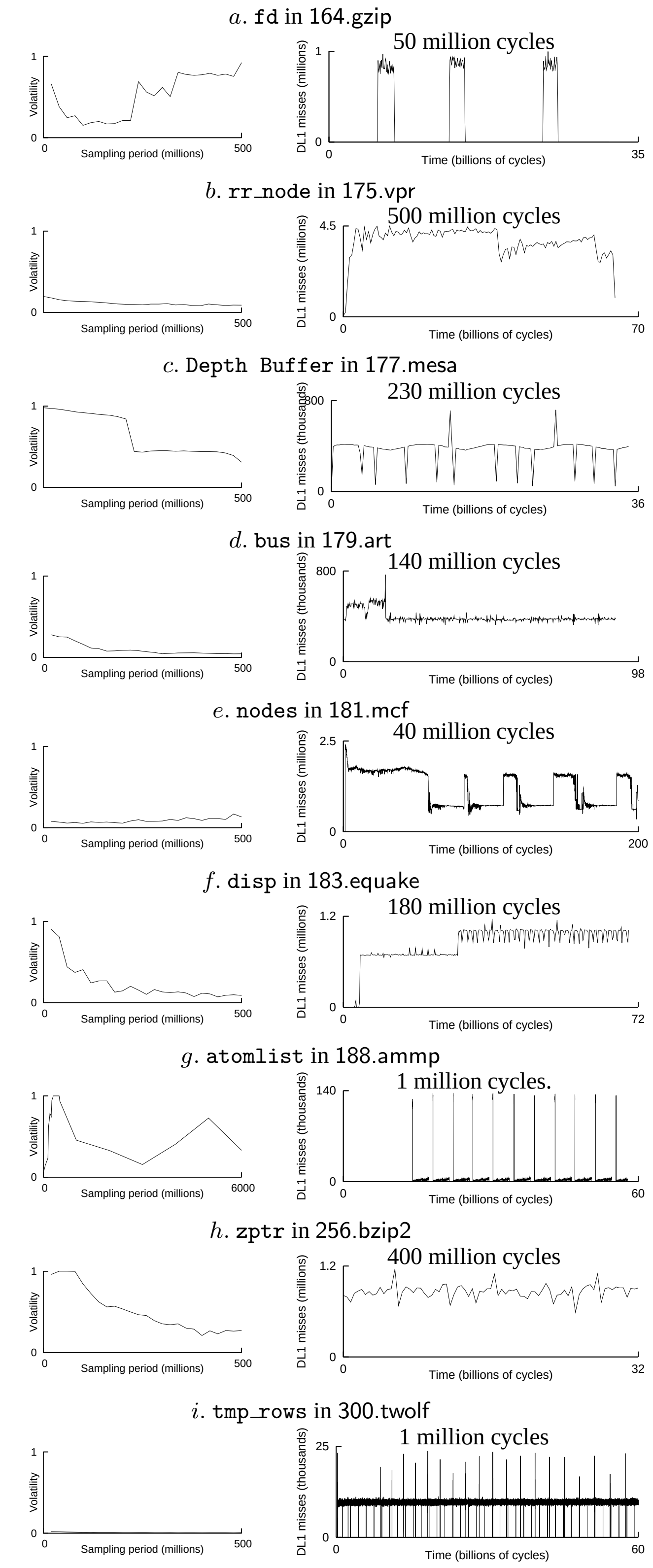

Figure 10. Volatility profiles of some major data structures in our applications (left), and the corresponding phase behavior at one low-volatility sampling period in the profile (right).

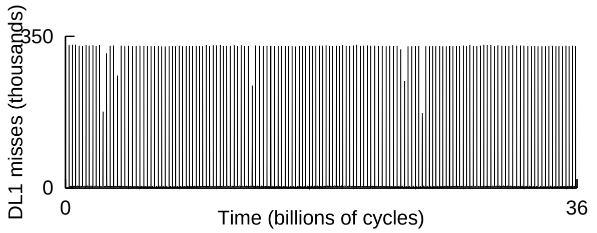

Figure 11. The phase behavior of 177.mesa at 10 million cycles. Compare with Figure 10*c*.

plications, along with the curve corresponding to a low-volatility sampling period in each.

While two applications in Figure 10 – 181.mcf and 300.twolf – show consistently low profiles so that an arbitrary selection is likely to highlight global phase behavior, the volatility profiles of the others show that the sampling period must be selected carefully. The graphs on the left-hand side in Figure 10 can be partitioned into two broad classes of applications: those with monotonically decreasing volatilities as sampling periods increase, and those where volatility sometimes increases. A monotonically decreasing volatility profile is easily explained by the natural damping effects of aggregation to ever-increasing sampling periods. Selecting sampling periods in these cases is as simple as setting a threshold on volatility. Cases where volatility sometimes increases with sampling period are more interesting.

At a high level an application consists of nests of loops that access different data structures in different ways. The access pattern of a given data structure in a given loop may contribute a component with a certain approximate period to the phase behavior of the data structure. Combining all the interacting periodic components corresponding to a data structure yields the overall phase behavior of that data structure. If all the components for a data structure have relatively low time periods and high frequencies, we expect aggregation at high sampling periods to smooth out their disparate periodic effects. If a stream contains a component with a substantial time period, however, we observe a steeply oscillating volatility profile, with troughs at factors and multiples of that time period.

Such streams with coarse-grained periods make it more difficult to select a sampling period, requiring volatility measurements at a large number of values in order to find good candidates. For example, if a stream is dominated by a a period of 7 million cycles, taking measurements at sampling period increments of 10-million could fail to identify a good sampling period. By the time we find low volatility (at a sampling period of 70 million cycles) we may have damped out all phase behavior. Understanding such interactions in application phase behavior is a challenge for future research. In the context of this study, finding a low-volatility sampling period required gradually refining volatility measurements for 177.mesa and 188.ammp. As a concrete example of this, Figure 11 shows the phase behavior seen for `Depth Buffer` in 177.mesa at a sampling rate of 10 million cycles. Comparing this curve with that in Figure 10*c* shows how widely dissimilar different a stream can look at different sampling periods, and how selecting a bad sampling period can occlude gradual periodic patterns. The global phase behavior seen in Figure 10*c* is only observable in a narrow window of sampling periods, from 200 to 300 million cycles.

4.4 Types of phase behavior at a good sampling period

Having studied the volatility profiles on the left of Figure 10, we now study the phase transitions of data structures in these applications. Across all our applications, different data structures share common phase transition points. As a result, we are able to focus on a single data structure for each application in Figure 10 (right column). These phase graphs are of three types:

1. No phase behavior past initialization. 179.art, 183.equake, 300.twolf show this pattern.
2. Simple phase behavior between a well defined set of phases with easily-discerned boundaries. Examples of such behavior are 164.gzip, 181.mcf and 188.ammp.
3. More complex curves with poorly defined phases and fuzzy phase boundaries. Our exemplars are 175.vpr, 177.mesa and 256.bzip2.

Categories 2 and 3 both contain applications with phase inversions, where a different data structure contributes the most cache misses in each phase. Figure 12 shows the phase behavior of the major data structures in those of our applications with such inversions – 164.gzip, 175.vpr and 181.mcf. Identifying such phase behavior can be useful in several areas, such as adaptively varying processor issue width or cache capacity [21, 2]. Combining these prior implementations with data structure decomposition and the correct sampling period can provide a more rigorous framework for more sophisticated decisions.

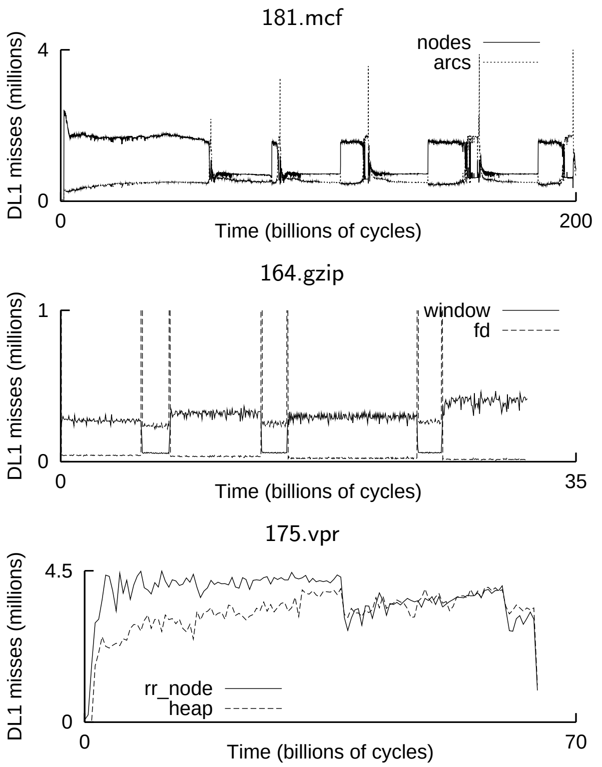

Figure 12. Applications with inversion: a different data structure contributes the most misses in each phase.

5. Conclusions and future work

In optimizing the performance of the memory hierarchy, architects and compiler writers have traditionally had very different views of application programs. Architects have usually treated the application as a black box and focussed on regularities in the overall address stream, while compiler writers and application programmers have focussed on identifying fine-grained optimization opportunities without access to detailed runtime information. In this paper, we combine the advantages of the two approaches by gathering runtime information and correlating it with program features - data structures and program phases - in a semi-automated way.

Our first contribution is a novel methodology for decomposing the address stream into multiple streams. Our methodology yields more detailed characterizations of applications that provide a richer view than the aggregate statistics of conventional methodologies. Applying it to twelve of the C SPEC2000 benchmarks is successful at highlighting and quantifying the variability in miss distributions and access patterns in the SPEC benchmark suite. It is also able to focus on the specific data structures that show unique behavior, such as a disproportionate number of memory stall cycles.

A second contribution of this study is a new framework to manage and understand application phase behavior at the right granularity. We show that data structure decomposition and sampling period selection are both important steps in studying an application's phase behavior, with significant impact on the final picture of the application that emerges. Understanding how sampling period influences phase behavior is complementary to recent work in detecting phase boundaries using code signatures [21, 11, 10]. One straightforward way to integrate it with online signature-based phase-detection techniques is by extending them to use variable sampling periods, determining the sampling period of an application offline and providing this information as a hint to hardware.

While this study focusses on C programs, our methodology and algorithms are applicable to other programming languages as well. In particular, they need only minor modifications to be applicable to garbage-collected runtimes – instrumentation in the garbage collector instead of the manual deallocation routine. Our methodology for selecting the right sampling period is loosely based on spectral analysis, and forms a general and rigorous approach to study phase behavior and consistently compare global phase behavior patterns across applications with seemingly different periodicities. One open problem is to improve this method to more gracefully identify sampling periods for applications with coarse-grained periodic behavior, and we believe more advanced ideas from Fourier analysis will be useful.

Acknowledgments

The design of DTrack was influenced by discussions with students in the CART and Speedway research groups. We also thank Michael Hind, Peter Sweeney and Kemal Ebcioglu for useful discussions about the relationship between sampling granularity and phase behavior. This research is based on work supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106, and by the NSF under instrumentation grant EIA-0303609 and grant CCR-0311829.

References

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 28th International Symposium on Microarchitecture*, Austin, TX, Dec. 1993.

[2] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 93–104, 2004.

[3] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Department of Computer Sciences, University of Wisconsin-Madison, June 1997.

[4] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[5] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 279–290, New York, NY, USA, 2002. ACM Press.

[6] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[7] S. Z. Guyer, D. A. Jiménez, and C. Lin. The C-Breeze compiler infrastructure. Technical Report TR 01-43, Dept. of Computer Sciences, University of Texas at Austin, November 2001.

[8] I. J. Haikala and P. H. Kutvonen. Split cache organizations. In *Performance '84: Proceedings of the Tenth International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 459–472. North-Holland, 1985.

[9] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, Dec. 1988.

[10] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[11] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[12] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, Oct. 1994.

[13] M. Martonosi, A. Gupta, and T. E. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 1–12, Newport, RI, June 1992.

[14] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, MA, Oct. 1996.

[15] P. Nagpurkar, M. Hind, C. Krintz, P. Sweeney, and V. Rajan. Online phase detection algorithms. In *Proceedings of the 4th annual international symposium on code generation and optimization*, March 2006.

[16] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[17] S. Rubin, R. Bodik, and T. M. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[18] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, San Jose, CA, Oct. 1998.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[21] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th International Symposium of Computer Architecture*, pages 336–347, June 2003.

[22] A. J. Smith. Second bibliography on cache memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[23] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.*, 37(5):562–573, 1988.

[24] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 132–144, June 2001.

[25] E. van der Deijl, G. Kanbier, O. Temam, and E. Granston. A cache visualization tool. *IEEE Computer*, pages 71–78, July 1997.