# Phase Analysis of Program Memory Behavior

Kartik K. Agaram  Stephen W. Keckler
Calvin Lin  Kathryn S. McKinley

Department of Computer Sciences
Tech Report TR2002-67
The University of Texas at Austin

**ABSTRACT**

This paper describes a methodology for measuring and analyzing a program's memory performance. This method can both identify phase behavior and analyze the behavior of individual data structures. Our approach uses a new tool that combines compile-time annotation of memory allocation sites with a detailed microprocessor simulator. Using this infrastructure, we decompose the complex access patterns of four SPEC-2000 benchmarks by phase and by data structure. We study how different data structures coexist in a common memory system and we distinguish data structures that miss because of external conflicts from those that miss because of poor intrinsic locality. These results provide a richer understanding of the application than those delivered by tools that simply aggregate memory behavior into a single miss-rate statistic. These results also suggest effective optimizations for each data structure and phase of an application. In various phases of the execution of the benchmarks we identify optimizations, such as data-structure specific caching, static layout transformations, software prefetching and streaming, that are likely to be most effective. We show that the set of effective optimizations vary by application and program execution phase.

# 1 Introduction

The memory hierarchy of computer systems has a substantial, if not dominant, effect on application performance. It is not surprising then that many researchers are attacking memory latency at all levels of computer systems. Architects reduce memory latency and increase bandwidth by implementing and optimizing hardware caching algorithms; compilers improve the effectiveness of caches through optimizations such as cache blocking, selective caching, and prefetching; and application writers often tune their applications to the size of the caches, either by hand or in an automated fashion [21, 5]. Tools for analyzing cache behavior include cache simulators, such as the sim-cache [3] simulator from the SimpleScalar tool suite, and hardware performance monitors [2]. These tools have an architectural point of view and do not describe performance problems in terms that are familiar to the programmer's or compiler's view of the program. Several program analysis tools [20, 8, 11, 14] come closer to this view and capture aggregate data structure, procedure, and/or loop nesting cache behavior for array programs. They do not, however, examine *phase behavior*—how data structure behavior varies over the execution of the program. These tools also do not analyze dynamically allocated pointer data structures, which are responsible for poor cache behavior in many programs.

This paper introduces *DTrack*, an automated tool that tracks a program's memory behavior in terms of individual data structures across the execution of the program. DTrack monitors accesses to the stack, heap and global segment, and categorizes each access to the specific data structure it belongs to. Each access is tracked through the cache memory hierarchy, and the system provides cache hierarchy statistics, such as access counts, hits, misses, and miss rates, on a per data structure basis. DTrack also reports the time-varying behavior of these statistics across the execution of the program. These statistics identify subtle (and not so subtle) program characteristics such as hot data structures, data structure interference in the cache hierarchy, and performance bottlenecks due to layout or access patterns of individual data structures. These new capabilities provide better insight into program behavior than prior tools that aggregate statistics across both data structures and time. As a result, DTrack identifies opportunities for hardware and software optimizations at many levels. This analysis will be more important in future architectures as memory latency increases and partitioned architectures become more common. For instance, the use of recently proposed hardware such as partitioned caches and reconfigurable caches [16, 10] calls for the data-structure phase analysis that DTrack provides.

DTrack includes two main components: a compile-time data structure analyzer and a microarchitectural simulator that gathers the runtime statistics for each data structure. The compile-time analyzer annotates all dynamic memory allocation call sites (e.g., *malloc*) and generates a map of the address space. The simulator reads the map and assigns each access, hit, and miss to the corresponding data structure. The simulator also reports the data in execution time intervals, thus capturing phase behavior in the program. To illustrate the capabilities of DTrack, we use it to analyze the behavior of four frequently missing SPEC-2000 benchmarks, each of which illustrates different data-structure based memory system performance. Our analysis suggests opportunities for optimization including: reducing interference between two hot data structures, data structure reorganization to improve locality, data structure partition sharing for partitioned caches, and data structures that should be streamed directly into the processor, bypassing the cache hierarchy.

The main contributions of this paper are a new methodology and supporting tools for combining information about program phase and individual data structure behavior, which yields new insights for architects, compiler writers, and application writers to use in memory system optimizations.

The remainder of this paper is organized as follows. Section 2 describes prior memory system analysis tools and details the differences between them and DTrack. Section 3 describes the components of DTrack, explains the mechanisms employed to minimize the invasiveness of the annotations, and mea-

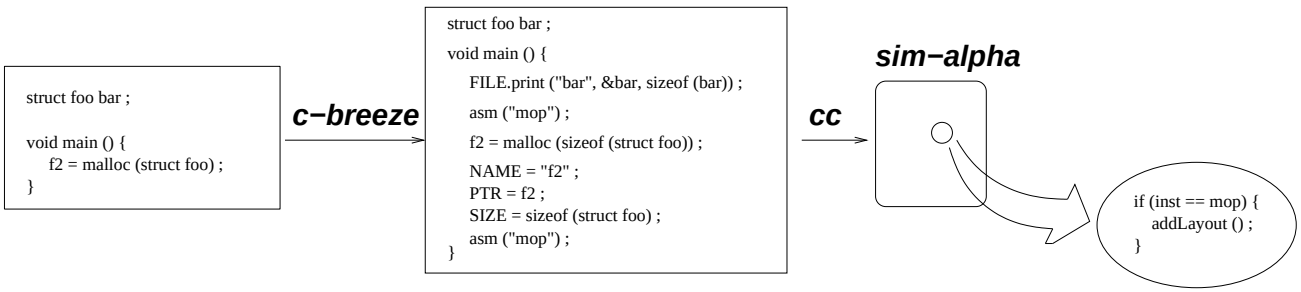


Figure 1: DTrack schematic.

sures both the invasiveness and the impact on simulation time. Section 4 demonstrates the capabilities of DTrack on the four benchmarks, highlighting the additional insight gained by the per data structure and time-based statistics. Section 5 discusses performance techniques motivated by the results and insights of Section 4. Finally, Section 6 summarizes and suggests further uses of DTrack.

## 2 Related Work

A common method of producing aggregate memory statistics is through simulation, of which we cite a few representative samples [1, 7, 19, 3]. More sophisticated cache memory behavior analysis tools have been developed [20, 8, 11, 12, 13, 14], and this section compares DTrack to this prior work. Our work differs from these tools in that we consider pointer data structures in addition to arrays, and show that aggregate statistics obscure possible optimization opportunities revealed by phase behavior. Of course, the increased detail comes at a cost of increased simulation time.

Most tools [8, 11, 12] have focused on aggregate data structure and procedure-level information for arrays. Lebeck et al. [8] and Martonosi et al. [11] present data structure and procedure level aggregate miss information, and classify misses as compulsory, capacity, and conflict. Both papers also present a number of software optimizations for improving cache performance. While these tools point users to the code and arrays that cause problems, they examine the behavior of an array within the context of a single procedure, resulting in two weaknesses. First, because they do not perform cross data structure analysis, it is not directly apparent from their aggregate data statistics which data structures interfere with themselves or with others. Second, since they do not perform cross-procedure analysis, optimizations chosen to improve performance of one array/procedure combination may diminish performance in another procedure. Finally, both tools handle only regular array-based data structures rather than pointer-based data structures. McKinley and Temam analyze the complementary dimension of inter-nest and intra-nest loop locality [13, 14], but again consider only arrays and aggregate information between loop nests.

The tool closest to ours is the Cache Visualization Tool [20] which demonstrates the time-varying behavior or arrays as they march through the cache. The graphical component of this tool colors cache lines according to their locality and misses by data structures, so the user can see which cache lines cause conflict misses. This level of detail supports analyzing a single loop nest at a time, whereas we compute and present data structure phase behavior for much longer periods. In addition, we also analyze pointer codes whose data is not well structured.

# 3 DTrack

This section describes DTrack, our tool for performing detailed analyses of applications from a memory system perspective. Figure 1 shows a schematic of the tool chain. DTrack consists of an extension to the C-Breeze C-to-C translator [6], forming the front-end, and an extension to a detailed microprocessor simulator called sim-alpha [4]. We use sim-alpha to associate memory system behavior, such as memory accesses and cache misses, with source-level data structures, and we use C-Breeze to instrument the application source so that the simulator can map addresses to application-level data structures.

## 3.1 Identifying Data Structures

Identifying the appropriate mapping from address to data structure is hard in the case of recursive data structures such as lists and trees. Scalar and array data structures are easily identified at compile-time because their allocations are coupled with their declarations. Pointer-based data structures, however, are allocated dynamically and often in a piecemeal fashion, so a method of relating dynamically allocated memory to individual data structures is necessary. We use a simple heuristic that considers all allocations from the same program location to belong to the same data structure. In general, this approach could fail to distinguish allocation to different instances of the same type, e.g. two distinct trees. For the SPEC benchmarks that we consider in this work, we find that the predominant data structures do not have multiple instances, and are usually allocated from a single location. In the rare case where multiple locations in the program allocate nodes to the same data structure, we manually coalesce nodes allocated at these locations. Other languages and benchmarks may require more elaborate heuristics.

## 3.2 Communicating Data Structure Layout to the Simulator

The goal of the source-level translator is to communicate to the simulator information about the extents of different data structures in simulated memory. In particular, the translator instruments the application to compute the name, address, and size of each data structure and provide them to the simulator. The simulator maintains this information in a tabular layout by address range. To minimize the invasiveness of the tool on the underlying application in different contexts, the translator uses two techniques to communicate layout information.

- For global variables, the names and addresses of variables are written to a predecided file. These file operations, though expensive, are one-time costs during program initialization that are amortized across all global variables.
- A more efficient solution is needed for heap-allocated variables, since the simulator needs to associate a data structure name with each dynamic allocation or deallocation. The translator inserts instrumentation code that stores the name (a numerical encoding of the function it is found in), the address of the allocated data, the size of the allocated memory, and the type of memory operation (allocation or deallocation). All of this information is stored in a predetermined set of variables so that the simulator can extract it from the application's memory during simulation. To reduce the impact of these extra variables, we simulate perfect memory for them, and never fetch them into the simulated cache.

In both cases, when the new additions to the layout are in place the front-end instructs the simulator to extract them by inserting a specific rarely used opcode. When the simulator encounters this opcode in the simulated instruction stream, it processes the instruction and then takes additional measures

to either read a file or import variables from the application's memory. With these mechanisms, the simulator is able to add or remove entries from the layout as memory is allocated or freed on the heap, to always maintain an accurate picture of the application's data structures. This knowledge of the current layout of the program throughout its execution enables it to collect statistics on a per data structure basis. The code fragment in Figure 1 illustrates the entire process. Two details merit attention:

- Stack variables would be expensive to instrument, since they are allocated and deallocated on every change in scope. We choose to treat the stack as a monolithic entity. Detect accesses to the stack requires no instrumentation by the translator, since it corresponds to a well-delineated region in memory. We show in the next section that combining stack variables does not reduce DTrack's effectiveness.

- All instrumentation is performed by transforming the application source so variables within pre-compiled libraries need to be handled specially. The basic idea is to assume that each library allocates exactly one data structure. Thus, when the simulator encounters an allocation inside of a library routine, it names the data structure by the name of the library routine. Our system does not track any global variables declared by library routines. We find such variables to be rare.

We measure the overhead introduced by DTrack in instrumenting dynamic allocations and deallocations, by comparing, for each of our benchmarks, the instruction counts executed both with and without instrumentation, to reach a specific point in the source code past initialization and including one to ten iterations of the top-level loop of the benchmark. This comparison shows that our instrumentation increases the instruction count of the benchmarks by a maximum of 0.3%.

### 3.3 Full Application Simulation

After the front-end instruments a program's sources, we compile them in the normal manner using cc on the alpha platform and simulate the resulting binary on sim-alpha. Details of the processor and memory configuration we simulate are presented in Table 1. The configuration we simulate is similar to the Alpha 21264 processor currently in the market. Previous research [15] has shown that, for a common program, an out-of-order processor provides a very different sequence of memory accesses to its caches, when compared to an in-order processor. We choose to performed detailed out-of-order simulation, so as to more accurately model the behavior of our benchmarks on contemporary computer systems. We modify the simulator to correctly process the instrumentation provided by the front-end, and to correctly classify all memory accesses based on them. Classifying all loads and stores in this manner and keeping track of statistics on a data structure basis slows down the simulation by an average of 60%.

In order to capture the important phase behavior in our benchmarks, we simulate them for 40 billion instructions each from the start. This large simulation effort is required to obtain accurate and comprehensive results on the phase behavior of these programs. Sherwood et al. [18, 17] recently developed a tool called SimPoint that breaks up a program's execution into slices, clusters execution slices on the basis of a code similarity metric, and predicts the slice in each cluster that is most representative with respect to the miss-rates and IPC (instructions per clock cycle). Simulating a small number of thin slices of the whole program and weighting them appropriately provides results with very low error. While this methodology has been validated with respect to aggregate statistics, it is not clear how much error it introduces when studying phase behavior. We choose, for this reason, to simulate large parts of the execution of all our benchmarks. To ensure that the first 40 billion

| Feature | Size/Value |
|---|---|
| Out-of-order Processor | |
| Fetch width | 4 |
| Decode width | 4 |
| Issue width | 4 |
| Int ALUs | 4 |
| Int multipliers | 4 |
| FP ALUs | 1 |
| FP multipliers | 1 |
| Branch predictor | Tournament, 1 KB x 1 KB local, 4 KB global, 4 KB choice |
| Memory Hierarchy | |
| Level 1 Data Cache (DL1) | 64 KB, blocksize 64 bytes, 2-way |
| DL1 latency | 3 cycles |
| Level 1 Instruction Cache (IL1) | 64 KB, blocksize 64 bytes, 2-way |
| IL1 latency | 1 cycle |
| Unified Level 2 Cache (L2) | 512 KB, blocksize 64 bytes, direct-mapped |
| L2 latency | 12 cycles |
| Translation Look-aside Buffers (TLBs) | 128 entries |
| Latency to DRAM | 62 cycles |

Table 1: Details of the simulated processor and memory hierarchy.

instructions provide representative results, we ran all the benchmarks to completion in an earlier experiment. Comparing those results with those of SimPoint, we find that while SimPoint predicts aggregate miss-rates with 0.0% error for three of the four benchmarks, it shows significant error in the mean miss-rate within a cluster, with an average of 11%, and as high as 47.7% for some clusters in our benchmarks.

To simulate large intervals of many billions of instructions, we partition each simulation into multiple runs and simulate them in parallel on a cluster of Linux workstations managed by Condor [9]. Each of these runs performs functional simulation (fast-forwards) to a specific point and then simulates a billion instructions. Different runs fast-forward different distances. The results from these staggered runs are post-processed offline to provide results for the entire simulation.

Our parallel approach introduces errors due to the cold caches that appear every billion instructions. Since each billion-instruction sample finds at least 10 million misses in the DL1 and 1 million misses in the L2, the error due to extra compulsory misses is a maximum of 512 misses in the DL1 and 8192 misses in the L2 in every billion instructions, which is an acceptable level of error.

# 4 Results

In this section, we use DTrack to examine the memory hierarchy behavior of four SPEC-2000 benchmarks, highlighting the types of insight gained first by the aggregate per data structure analysis and then by temporal analyses of phase behavior. We then use these insights to suggest and explore static and dynamic methods of improving performance in Section 5. We expect that the same analysis will be useful beyond our example benchmarks, and could, in fact, suggest additional performance optimizations tailored to each particular application.

| | DL1 | | | L2 | | |
|---|---|---|---|---|---|---|
| Benchmark | Accesses | Misses | Miss-rate | Misses | Miss-rate | IPC |
| ammp | 14.9G | 1.7G | 11.1% | 0.6G | 36.7% | 0.82 |
| art | 20.3G | 7.1G | 34.9% | 4.6G | 64.2% | 0.54 |
| equake | 19.3G | 2.9G | 14.1% | 0.8G | 29.4% | 0.58 |
| mcf | 18.9G | 8.8G | 46.4% | 4.0G | 44.4% | 0.22 |

Table 2: Aggregate memory hierarchy behavior.

### 4.1 Aggregate Data Structure Analysis

Table 2 summarizes the aggregate behavior of four example benchmarks taken from the SPEC-2000 suite. Each benchmark was run for 40 billion instructions. The high DL1 and L2 miss rates indicate that these applications are memory intensive. Figure 2 shows the aggregate per data structure behavior of these benchmarks produced by DTrack. The five most important dynamically allocated data structures (ordered by miss count) are shown for each benchmark, along with the bars for stack accesses and all remaining accesses ("Others").

In the level-1 data cache (DL1) more than 60% of the misses result from references to a single data structure, while three of the benchmarks show a significant second data structure (middle row). The data structures that have the most misses have significant access counts (top row). In art, 10% of all DL1 accesses are data structures that almost always miss. More than half of the critical data structures have higher-than-aggregate DL1 miss-rates, often significantly higher.

To examine each application in further detail, we used the results of DTrack as a guide to the source code. Table 3 summarizes the top data structures from Figure 2, showing the total size of the structure, the size of each element, and the way in which the structure is accessed (type). Note that 2-D arrays that are implemented as arrays of arrays are shown as separate data structures. For example, `bu` is an array of pointers, each of which point to data arrays that are represented by `bu[]`. The type indicates the whether the data structure is accessed in a regular fashion (array), or sometimes in a regular and other times in an irregular fashion. In the paragraphs that follow, we describe further details of the applications and their most significant data structures.

**ammp** models the molecular dynamics of a protein in water. It tracks the motion of a set of approximately 10,000 atoms from an initial configuration by repeatedly solving a system of differential equations for each atom. The set of atoms to be modeled is maintained as a linked list that is repeatedly traversed as forces and velocities are computed. To model interactions, each atom maintains an array of 200 neighboring atoms. This array of neighboring atoms must be periodically recomputed for each atom, as it moves through the space. In addition to the list of atoms, the program maintains several auxiliary lists that contain information about bonds between atoms, angle computations, torsional forces and tetrahedral structures. The length of each of these lists is roughly proportional to the number of atoms. The interlocking nature of these different data structures makes it difficult to statically determine the critical ones.

Using DTrack, we determine that the list of atoms (`atoms`) causes the most misses, more than four times that of any other data structure. The large size of the data structure (80 MB) and the common pattern of traversing it from end to end contributes to its poor cache behavior, causing a large number of capacity misses. Surprisingly, the auxiliary lists of physical relationships between atoms, such as angles and torsion, create few misses in the memory hierarchy. Instead, almost 24% of all misses are caused by 4 temporary arrays which are repeatedly re-allocated during during program execution and used in only one of the 150 functions in the call-graph of ammp. Determining that these data

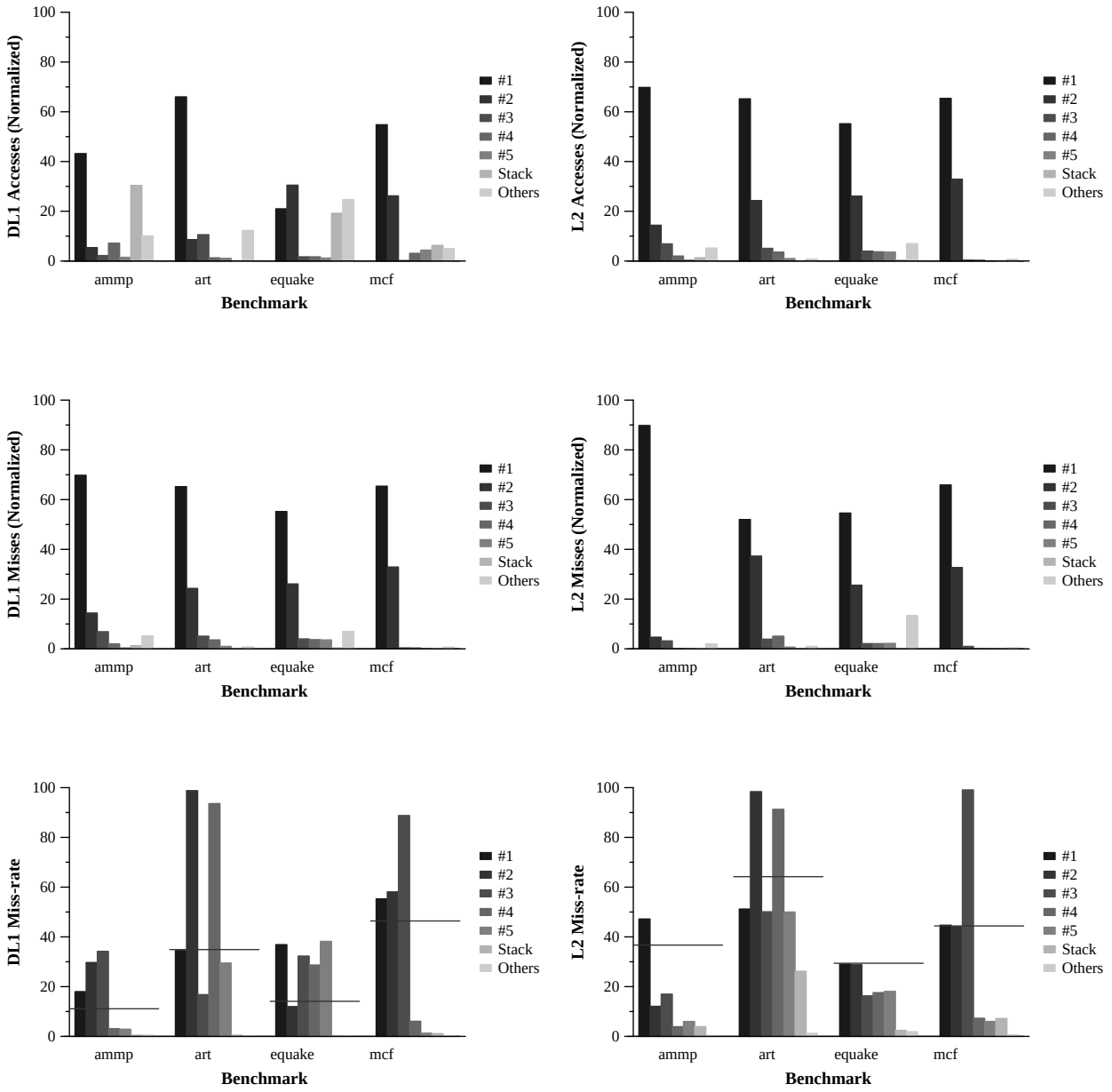


Figure 2: Decomposition of DL1 and L2 behavior by data structure, showing accesses, misses, and miss rate. The horizontal bars on the bottom graphs indicate the total aggregate miss rate on all of the data structures.

| Data Structure | Type | Access Pattern | Size (KB) | Element size (bytes) |
|---|---|---|---|---|
| Benchmark: ammp | | | | |
| #1: atoms | Pointer | Both | 41322 | 2208 |
| #2: nodelist | Array | Regular | 76 | 232 |
| #3: atomlist | Array | Regular | 4342 | 232 |
| #4: vector | Array | Regular | 599 | 8 |
| #5: atomall | Array | Regular | 150 | 8 |
| Benchmark: art | | | | |
| #1: f1_layer | Array | Regular | 625 | 64 |
| #2: bu[] | Array | Regular | 859 | 8 |
| #3: bu | Array | Regular | 78 | 8 |
| #4: td[] | Array | Regular | 859 | 8 |
| #5: f1_layer[].I | Array | Regular | 156 | 8 |
| Benchmark: equake | | | | |
| #1: K | Array | Regular | 22399 | 8 |
| #2: disp | Array | Regular | 2828 | 8 |
| #3: V23 | Array | Regular | 943 | 8 |
| #4: C23 | Array | Regular | 943 | 8 |
| #5: M23 | Array | Regular | 943 | 8 |
| Benchmark: mcf | | | | |
| #1: nodes | Array | Both | 7071 | 120 |
| #2: arcs | Array | Irregular | 188416 | 64 |
| #3: dummy_arcs | Array | Irregular | 3771 | 120 |
| #4: basket | Array | Regular | 3.13 | 8 |
| #5: perm | Array | Regular | 3.13 | 8 |

Table 3: The most critical data structures by miss count.

structures are more critical than the persistent auxiliary arrays of similar length would be difficult without DTrack, which highlights them in an automated manner.

**art** performs image recognition using an unsupervised neural network classifier. This benchmark consists of regular loops traversing large 2-dimensional arrays. The major data structures, shown in Figure 2, are the array of neurons (`f1_layer`) and the arrays of bottom-up and top-down weights, `bu` and `td`. These two-dimensional arrays are organized as arrays of arrays, and fewer misses are due to the array of row pointers than from the subsidiary arrays holding the data. Further examination of the data structures profiled by DTrack shows that each iteration of the inner loop of art accesses 1-2 specific fields in the array of neurons.

This pattern suggests field-splitting - breaking up the array of neuron data structures into smaller arrays, each containing fields of the neuron. This optimization will eliminate fetching of unused neuron fields.

**equake** simulates the propagation of seismic waves in large valleys, determining the history of ground motion during an earthquake. It uses a finite-element computation on an unstructured grid topology, which involves regular traversal of large 2-D and 3-D arrays with access patterns similar to **art**. DTrack shows that the most commonly missing data structures are portions of two large 3-D arrays, `K` and `disp`. The innermost loop repeatedly multiplies the matrices in `K` with the corresponding vectors of a

matrix in `disp`. `K` is never written after initialization, while `disp` is frequently modified. The remaining 3 top data structures are part of a group of 5 data structures that are accessed in interleaved fashion to update an element in `disp`. `K` displays a miss rate of 37%, far exceeding the aggregate miss rate of 14.1%, while `C23`, `M23` and `V23` have miss rates exceeding 28%.

**mcf** implements the network simplex algorithm to minimize the number of vessels required in a fleet to traverse a graph of destinations with fixed arrival/departure schedules and preplanned routes. The principle data structures are the `nodes` and `arcs` shown in Figure 2, which collectively represent the graph of destinations. Each node contains a linked list of pointers to incoming and outgoing arcs, and each arc contains pointers to the nodes it connects. Each node also contains pointers to its parent and linked lists of children and siblings. In certain phases of the program nodes are accessed in regular order, corresponding to a depth-first search of a subtree of the graph, while in others node access is irregular as the nodes are accessed in the sequence of the arcs that they are connected to. The array of arcs is accessed in sequential order. Some loops insert a new arc at the beginning of the arc array, which triggers accesses that employ recursive doubling of the array indices ($2 * n$ or $2 * n + 1$) until an empty arc position is found. Other loops traverse the arcs and select a subset, placing them into a temporary buffer where they are sorted using a quicksort algorithm. The unstructured access patterns to `nodes` (850KB) and `arcs` (12MB) results in DL1 miss rates for each that exceed the aggregate miss rate.

### 4.2 Temporal Analysis

We now examine the behavior of the data structures over the execution time of the applications. DTrack exposes phase behavior on a data structure basis and shows correlation between different data structures within the phases.

**ammp:** As shown in the previous section, the cache misses in ammp are caused primarily by its linked list of atoms, and to a lesser extent by various temporary arrays based on this list and the 3-D grid that models the space around the atoms. Figure 3 plots the number of DL1 misses per 0.5 billion cycles for each of these data structures. The graph shows 110 time steps simulated by the outermost loop. The number of misses to the linked list of atoms and to the arrays based on it peaks approximately every 4 billion cycles. These 4 billion cycle periods correspond exactly with 10-12 iterations of the outermost loop. However, the time spent by the iterations within each period is not uniform. While most of the iterations take comparitively less time to execute, one iteration in every 10-12 executes a function called `mm_fv_update_nonbon`, which updates the neighbors for each atom. Over 80% of the time in each period is spent within this function. This function is solely responsible for the misses to 4 of the 5 critical data structures in ammp, and the traversals in the neighbor pointers in the fifth and most important data structure. Thus, DTrack is able to highlight in an automated fashion that optimization efforts should be focussed on the traversal of the neighbor array.

Figure 4 shows a magnified section of Figure 3 to demonstrate how a temporal analysis without data structure decomposition can miss important aspects of the program memory behavior. During a single phase, the miss counts for both `nodelist` and `vector` data structures initially rise, but the miss count for `vector` soon falls in the latter part of the phase. Tools that simply measure the phase behavior in aggregate miss count without separating the different data structures would observe a much flatter curve, and not expose the more dramatic shift in the fraction of misses contributed by each.

**mcf:** The execution of mcf is composed of alternating phases that perform an iteration of the simplex algorithm followed by insertion of new arcs into the graph. Figure 5 shows the DL1 miss count per 0.5 billion cycles for the top five data structures across the execution of the entire program on the

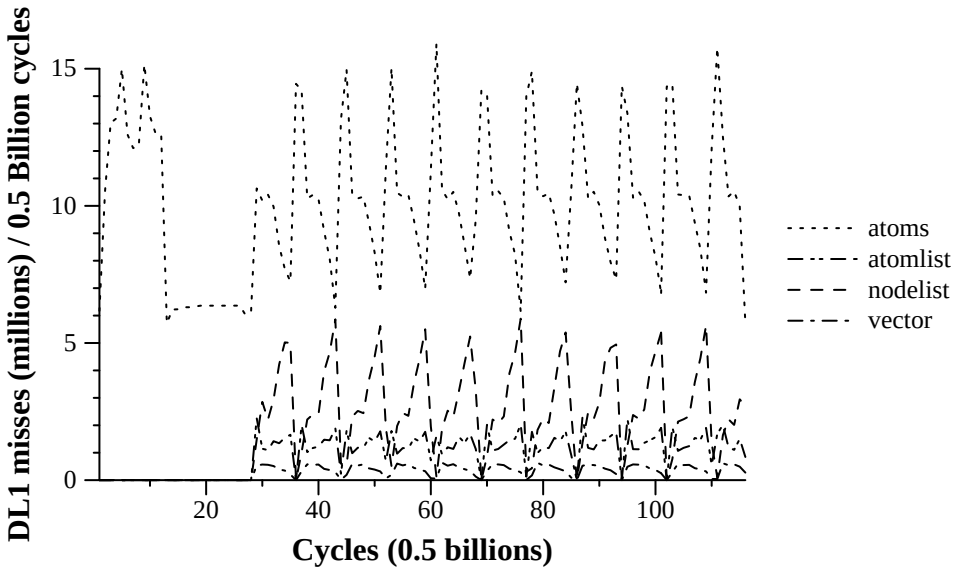


Figure 3: Misses by data structure per 0.5 billion cycles in ammp.

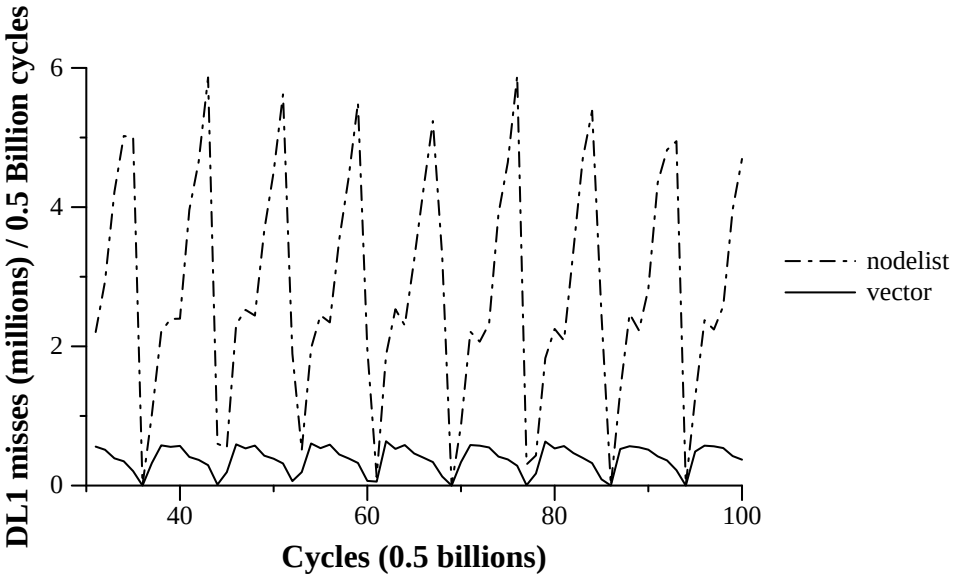


Figure 4: Magnified view of Figure 3.

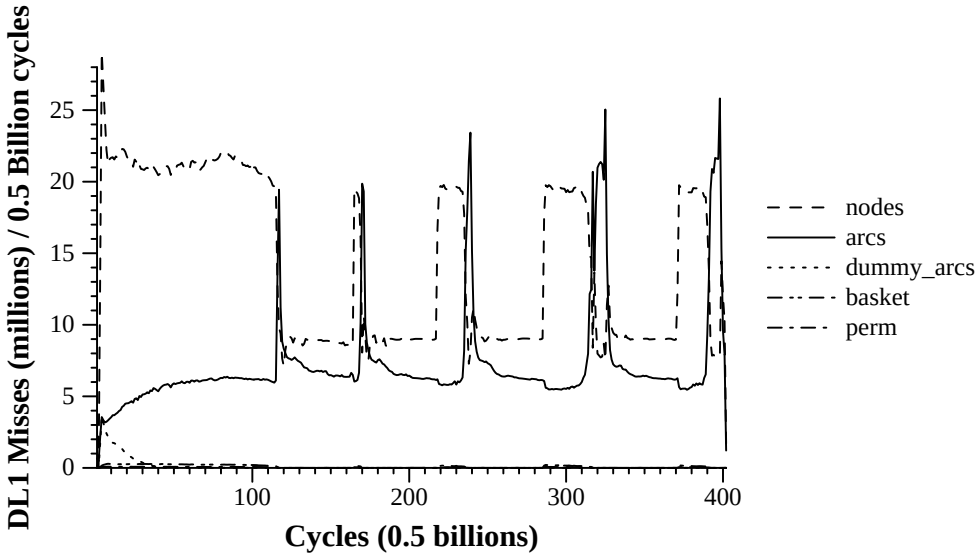


Figure 5: Misses by data structure per 0.5 billion cycles in mcf.

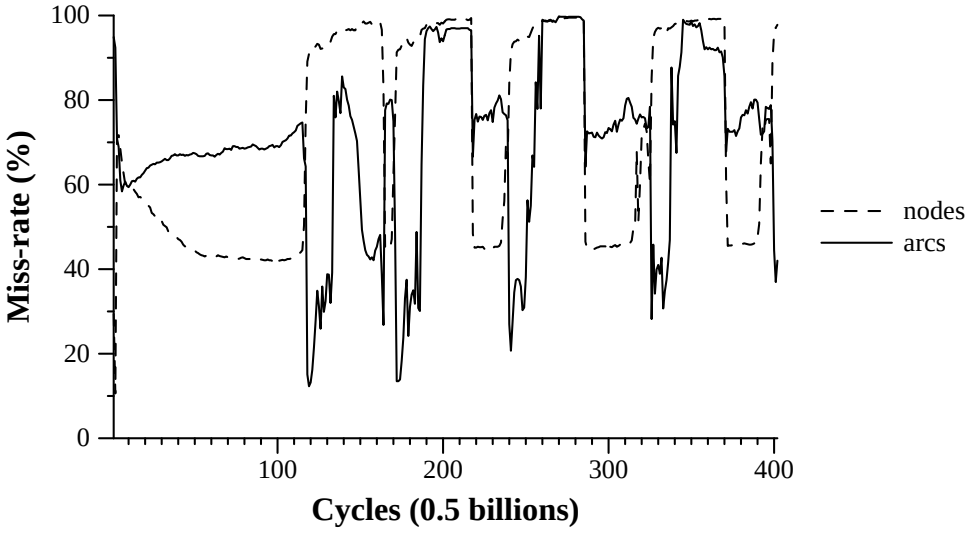


Figure 6: Miss-rates of the node and arc arrays in mcf.

reference input set. Misses to `nodes` and `arcs` tend to track one another except for spikes in the `arcs` curve. These peaks in misses to the arc array correspond to the start of the simplex phase in which a subset of the arcs are selected and sorted. The end of the simplex phase corresponds with the dramatic increase in misses to `nodes`. These misses correspond to the arc insertion phase in which the arc array is traversed sequentially with a stride of three. This phase periodically inserts new arcs into the array, causing some irregular access to the array and the secondary spikes in the miss count for `arcs`. In conjunction with the mostly regular accesses to `arcs` is the near random accesses to the nodes that the arcs connect. Figure 6 shows the corresponding miss rate for each sample point. The spikes in the miss rate for `nodes` occur at the same time as the spikes in the miss count.

**art and equake:** As shown in Figures 7 and 8, these two benchmarks show similar behavior characterized by only a single transition in memory system behavior, rather than the repeated phases in ammp and mcf. Note, however, that the transition occurs at between 15 and 30 billion cycles, points well beyond the simulation intervals traditionally picked by most researchers, prior to the recent development of SimPoint [18].

# 5 Memory Optimization Opportunities

By collecting statistics of the memory hierarchy behavior across the axes of data structures and time, DTrack exposes opportunities in the hardware and software to improve performance, which would otherwise be less obvious or invisible. The aggregate statistics on a data-structure basis alone immediately displays the data structures which have the poorest caching characteristics. As shown in Table 3, these are not always the largest data structures. This data quickly focuses the attention of the application programmer on the major bottlenecks in performance, and shows architects and compiler writers the areas that would most benefit from new caching strategies. The further detail provided by the time-based statistics exposes the structure and phases of the application, and indicates how a given data structure is accessed in different ways and frequencies over time. These observations inspire application, compile-time, and runtime optimizations that could exploit the time-varying access patterns to the critical data structures. The remainder of this section describes specific optimization opportunities exposed by DTrack.

**Data structure reorganization:** As shown in Figure 2, the bottom-up weight matrix `bu` in art accounts for a large number of misses and a particularly high miss rate in both the DL1 and L2 caches. Based on these statistics, we examined the references in the source code and discovered that this matrix is organized in row-major order, but accessed in column-major order. Since the rows are too large to fit in the cache, accesses to this matrix have neither spatial nor temporal locality. By transposing the matrix, we reduced its miss rate in the DL1 cache from 100% to 12.5% and improved overall IPC by 34% from 0.47 to 0.63. Not surprisingly, the miss rate shows that one in eight contiguous 8-byte accesses causes a miss in the DL1 with with 64-byte blocks.

**Avoiding caching conflicts:** Figure 2 shows that `Equake`'s second data structure `disp` is referenced more frequently than `K`, yet incurs fewer misses, indicating the possibility of cross-data structure conflicts. This suggests that a technique to partition the cache between the data structures would be useful. A more comprehensive way to detect such conflict is under development.

**Selecting data structure access method:** The aggregate data of Figure 2 shows extremely high miss rates in the DL1 and L2 caches (approaching 100%) for two data structures in art. Such poor

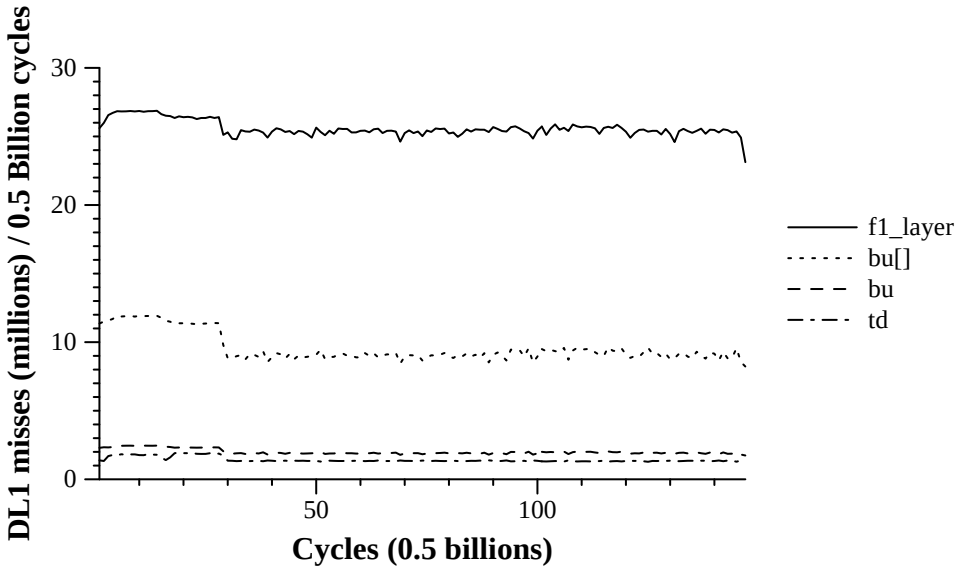


Figure 7: Misses by data structure per 0.5 billion cycles in art.

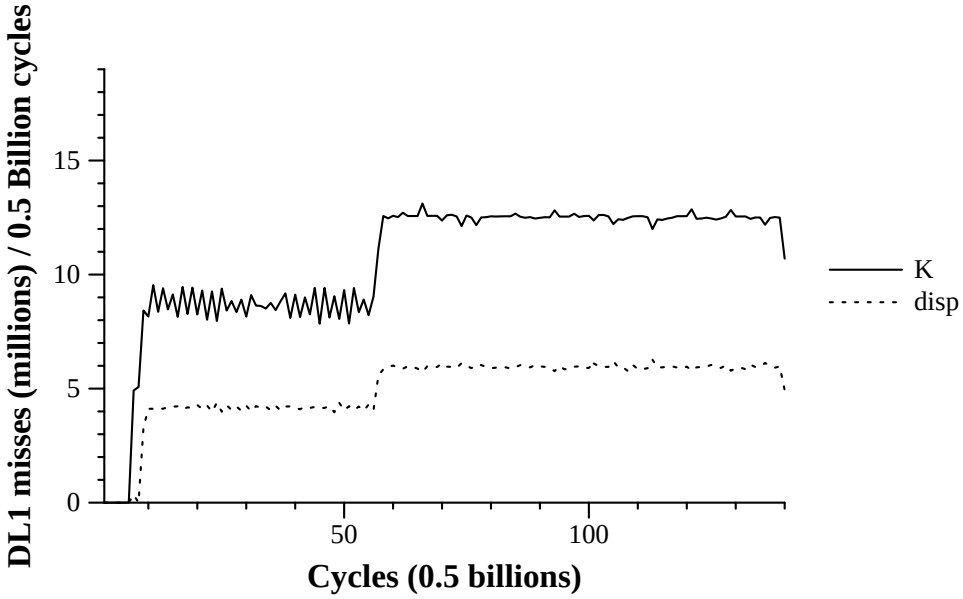


Figure 8: Misses by data structure per 0.5 billion cycles in equake.

cache behavior indicates that little spatial or temporal locality may exist and that accesses to these data structures would be better served by bypassing the cache entirely and avoiding pollution of the cache for the data structures that exhibit better locality. The temporal analysis shows that this behavior also manifests itself within a single data structure across the phases of a program. For example, Figure 5 shows that the `nodes` data structure goes through periods of high and low miss counts and rates. Based on this observation, we discovered that `nodes` is accessed in two different ways within the program. During periods of low misses, `nodes` is traversed as a list with good spatial locality. During periods of high misses, `nodes` is referenced indirectly from a traversal of `arcs`. The accesses to `nodes` during this phase is essentially random an results in poor locality. These results suggest that the references to `nodes` could be cached during one phase, and uncached in the other. Other access optimizations such as streaming and prefetching can be selected on a data structure basis or across phases within a data structure based on the results taken from DTrack.

**Cache reconfiguration:** Architectures that propose reconfiguration of cache organization and policies are now starting to emerge [16, 10]. The first steps will likely build reconfiguration into existing structures, such as cache partitioning based on set-associativity or adjusting effective cache line size by modifying the fetch policy. Future reconfigurations may include data structure specific caching. With its time and data structure statistics, DTrack can help determine when (between programs, between phases) and how to reconfigure. For example, programs such as mcf have different data structures that dominate the cache during different phases, while art and equake are very regular in the behavior. For those programs that have distinct phases, DTrack's identification of the per data structure behavior across phases enables exploration of the space of possible configurations and examination the benefits and drawbacks of dynamic reconfiguration.

# 6 Conclusions

Memory latency continues to determine the performance of many applications. Previous memory analysis tools detect troublesome array data structures based on aggregate misses. Our work combines this aggregate memory behavior with phase behavior analysis, and considers dynamic pointer data structures in addition to arrays. We use four programs to illustrate the utility of our methodology. Our phase analysis reveals how data structure misses vary over time and how data structures interact in the cache. This detailed information points to a number of application, compiler, and architectural optimizations that are not apparent from aggregate data.

Future extensions to DTrack will include miss types classification (conflict, capacity) on a data structure and phase basis. We will also enhance it to detect spatial and temporal locality in the data structure reference streams. Finally, we will use the results of these analyses to drive and evaluate the performance optimizations for both the compiler and architecture.

# 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*, pages 139–152, Austin, TX, Dec. 1993.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all

the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 1–14, October 1997.

[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] 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.

[5] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[6] 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.

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

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

[9] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[10] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[11] 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.

[12] M. Martonosi, A. Gupta, and T. E. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 248–259, Santa Clara, CA, May 1993.

[13] 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.

[14] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999.

[15] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[16] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[17] 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.

[18] 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.

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

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

[21] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Supercomputing Conference (SC98)*, pages 1–27, 1998.