The Memory Behavior of Data Structures in C SPEC CPU2000 Benchmarks

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

ABSTRACT

As a result of application and computer system design trends, the memory system continues to exert a dominant influence on program performance. While understanding how complex applications use complex computer systems is important to hardware, software, and benchmark designers, this is a difficult task. This paper demonstrates a system called DTrack that provides deeper insight into how an application uses the memory system. DTrack classifies memory accesses on a per data structure basis, enabling analysis of the regularity and locality of these program components. Applying the DTrack methodology to 12 C SPEC CPU2000 benchmarks, we demonstrate that this classification reveals data structure interactions that remain obscured with traditional aggregation-based analysis methods. Our characterization reveals the degree of diversity in memory behavior among the benchmarks within this suite, and we discuss how these insights can be used by system and application designers to achieve high performance through memory system and compiler improvements.

1. Introduction

A computer system's memory hierarchy has a substantial, if not dominant, effect on application 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. While many tools have been developed to analyze memory behavior [18, 9, 10, 11], none give insight into the behavior of individual data structures within a program. This paper describes our system - *DTrack* - which gathers memory system statistics on a per data structure basis, to help identify those data structures that have the strongest influence on performance and to offer insight into their size and access regularity.

DTrack consists of a C-to-C compiler that automatically instruments variable allocations in programs and a detailed timing simulator that consumes this instrumentation. This combination yields a tool that generates data profiles - detailed breakdowns of cache misses by the different high-level data structures in the source code. Given the data profile, we then manually combine it with a conventional code profile to determine the dominant access patterns for each data structure. Applying our methodology to 12 of the 15 C SPEC CPU2000 benchmarks, we demonstrate that our approach is able to partition seemingly irregular access patterns into streams that are more regular and easier to understand. Our characterization reveals important patterns in the distribution of cache misses in the major data structures of programs, and highlights unique behaviors in specific SPEC benchmarks. We demonstrate the usefulness of our tool by means of two case studies, showing sophisticated design decisions that a computer architect may face and the way that DTrack helps make these decisions.

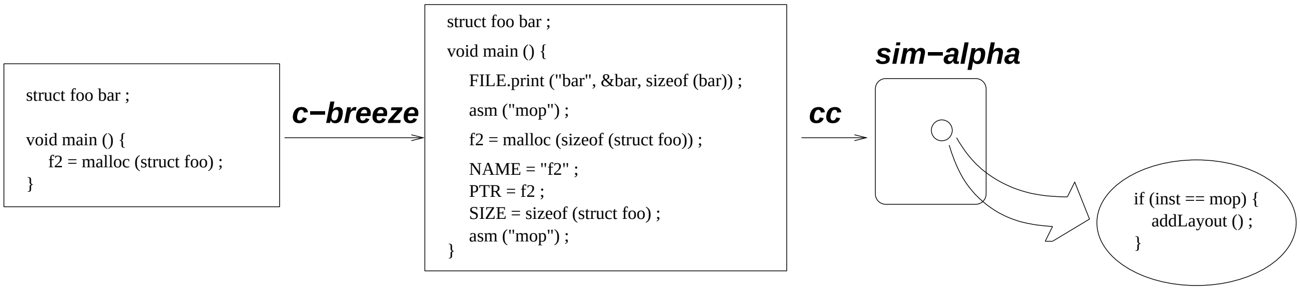
The remainder of this paper is organized as follows. Section 2 distinguishes our work from prior memory system analysis studies and tools. Section 3 describes our methodology, explains the mechanisms employed to minimize the invasiveness of the instrumentation we add to the benchmarks, and quantitatively measures both the invasiveness and the impact of our modifications on simulation time. Section 4 presents experimental results, highlighting the additional insight on dominanant access patterns gained by our detailed statistics, and illustrating how this rich picture of application behavior can be used from the perspective of a computer architect. 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, 8, 15, 2]. 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. DTrack provides a technique to connect these hardware events to high-level structures in the application source code.

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. [10] which operated on loop nests in Fortran programs. Similarly, Lebeck et al. [9] present data structure and procedure level aggreagate 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, and do not consider the question of whether data structures interfere with themselves or with others.

McKinley and Temam analyze the complementary dimension



```
struct foo bar ;

void main () {
    f2 = malloc (struct foo) ;
}
```

```
struct foo bar ;
void main () {
    FILE.print ("bar", &bar, sizeof (bar)) ;
    asm ("mop") ;
    f2 = malloc (sizeof (struct foo)) ;
    NAME = "f2" ;
    PTR = f2 ;
    SIZE = sizeof (struct foo) ;
    asm ("mop") ;
}
```

```
if (inst == mop) {
    addLayout () ;
}
```

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

of inter-nest and intra-nest loop locality [11], but again consider only arrays and aggregate information between loop nests. Finally, the Cache Visualization Tool [18] 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. Seidl and Zorn [14] 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. Finally, Chilimbi et al. [3, 13] 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.

3. Methodology

This section describes our methodology for performing detailed analyses of applications from a memory system perspective. We describe techniques to map addresses to data structures while minimizing the degree to which we perturb underlying application behavior. We also describe the machine configuration we simulate in our characterization, the simulation intervals we choose, and the aggregate statistics for our benchmarks that can be gleaned from conventional tools.

3.1. Mapping addresses to data structures

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 [6] C-to-C compiler. We then simulate them on a modified version of the sim-alpha [5] 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.

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

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 SPEC CPU2000 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,

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

Table 1: The benchmarks we use and their aggregate memory hierarchy behavior

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 will show that misses to the stack are generally negligible. In combination these techniques to instrument the global segment, heap and stack limit the perturbance due to our instrumentation to less than 0.6% of the instruction count 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.

3.2. Benchmarks, inputs and simulation intervals

This paper presents a characterization of 12 of the 15 C benchmarks in the SPEC CPU2000 benchmark suite. Table 1 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 SPEC CPU2000 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.

For each of our benchmarks we simulate a run using the designated ref input set. We demarcate the end of initialization by a special opcode using the techniques outlined previously in this section, and perform fast functional simulation until we reach this opcode. Thereafter we perform detailed timing simulation for 500 million instructions. Prior results over the entire simulated execution of 9 of our 12 benchmarks confirm that these simulated intervals are representative, with one exception: 181.mcf contains 2 separate phases that alternately stress the `nodes` and `arcs` data structures. We simulate only one of them due to lack of simulation time. As a result, while our results correctly reflect the major data structures in 181.mcf, they tend to under-estimate the importance of `arcs`.

4. Results

This section presents 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.

4.1. Data profiles and distributions

DTrack generates data profiles. Figure 3 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 3.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 3.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 3.a and 3.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.

4.2. 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. Table 2 now summarizes the high-level details of these data structures. For each benchmark, we show the name of these data structures as used in the source

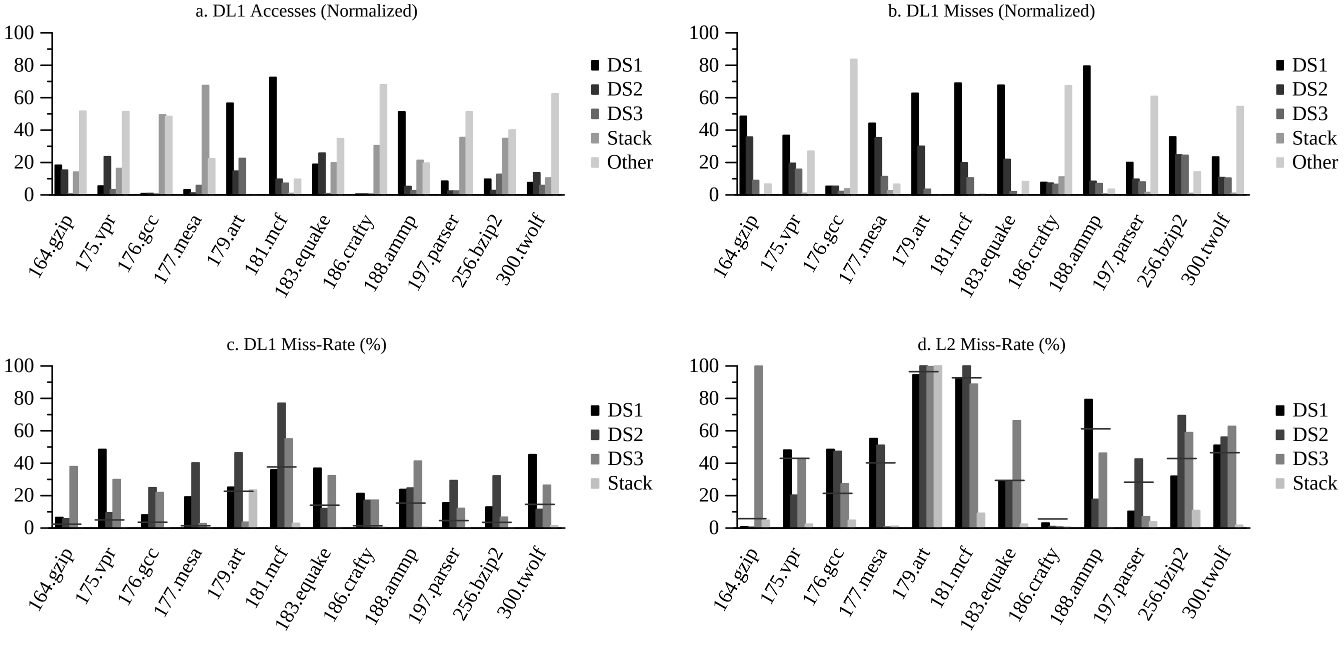


Figure 3: Decomposition of DL1 and L2 behavior by data structure. Horizontal lines in the miss-rate graphs indicate the aggregate miss rate for each benchmark across all data structures. L2 misses show similar trends to DL1 misses.

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.

Table 2 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.

While the regular applications 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 [12, 4] often cannot detect, or finally because we access 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.

4.3. Case study: Data structure criticality

Our first case study concerns criticality of memory reference. Several recent studies have showed that not all cache misses are equally important as measured in the amount of latency that they expose to the processor [17]. 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 [16]. 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 the neural-network simulation of 179.art; the array of top-down weights 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 – the above loop is the only major access pattern not shared with the dual array of bottom-up weights bus. The second data structure that we observe causing a disproportionate number of stalls is the variable search in

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[]→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

Table 2: Descriptions of the major data structures in Figure 3. Information on each benchmark for each major data structure: container type, access pattern, container and element size.

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 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.

4.4. Case study: Competition for caches

Where Figures 3.a and 3.b show the distribution of accesses to the DL1 and L2, Figures 3.c and 3.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

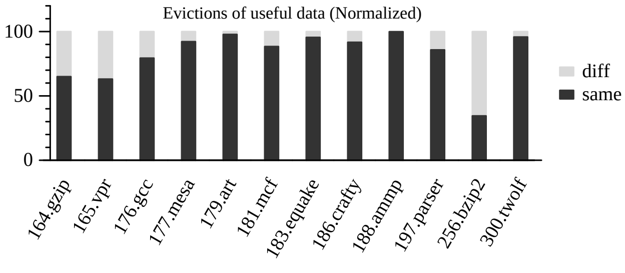


Figure 4: Breakdown of premature evictions: does the evicter belong to the same or a different (diff) data structure?

structures 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 [7] with static mapping policies assigning each data structure to a specific cache partition. Figure 4 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.

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 in a semi-automated way. The resulting methodology for decomposing the address stream into multiple streams yields more detailed characterizations of applications that provide a richer view than the aggregate statistics of conventional methodologies. Applying it to 12 of the C SPEC CPU2000 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.

Future work on this project continues along two major directions: extending our application analysis to a study of phase behavior, and using our insights in the design of a novel prefetching system that uses a combination of software hints and hardware prefetching to allow extremely early prefetching without any possibility of cache pollution. Preliminary results are promising and provide further evidence that the DTrack methodology highlights underlying regularities in application behavior, regularities that should prove useful in compiler, hardware and benchmark design.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[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] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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