Locating Cache Performance Bottlenecks Using Data Profiling

by

Aleksey Pesterev

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author . . . .
Department of Electrical Engineering and Computer Science
January 29, 2010

Certified by . . . . [illegible] . . . .
Robert T. Morris
Associate Professor
Thesis Supervisor

Accepted by . . . . [illegible] . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Locating Cache Performance Bottlenecks Using Data Profiling

by

Aleksey Pesterev

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Effective use of CPU data caches is critical to good performance, but poor cache use patterns are often hard to spot using existing execution profiling tools. Typical profilers attribute costs to specific code locations. The costs due to frequent cache misses on a given piece of data, however, may be spread over instructions throughout the application. The resulting individually small costs at a large number of instructions can easily appear insignificant in a code profiler's output.

DProf helps programmers understand cache miss costs by attributing misses to *data types* instead of code. Associating cache misses with data helps programmers locate data structures that experience misses in many places in the application's code. DProf introduces a number of new *views* of cache miss data, including a *data profile*, which reports the data types with the most cache misses, and a *data flow graph*, which summarizes how objects of a given type are accessed throughout their lifetime, and which accesses incur expensive cross-CPU cache loads. We present two case studies of using DProf to find and fix cache performance bottlenecks in Linux. The improvements provide a 16–57% throughput improvement on a range of memcached and Apache workloads.

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to have worked on this project with Nickolai Zeldovich and my thesis advisor, Robert Morris. They were both a great source of ideas and without either of their advice I would not have been able to complete this work. I also thank Frans Kaashoek for his encouragement and motivation.

This research was partially supported by a MathWorks Fellowship and the National Science Foundation.

Chapter 1

Introduction

Processors can consume data much faster than off-memory can supply it. While on-chip caches and prefetching can bridge some of this gap, it is nevertheless the case that programs often spend a significant fraction of their run-time stalled waiting for memory. Multicore chips will likely make this problem worse, both because they may increase total demand for data and because they allow for the possibility of cache misses due to contention over shared data. Understanding why software is suffering cache misses is often a pre-requisite to being able to improve the software's performance.

Following Hennessy and Patterson [12], the reasons for cache misses can be classified as follows. **Compulsory** misses are those taken the very first time a memory location is read. **True sharing** misses are those taken on core X as a result of a write from core Y invalidating data needed on X. **False sharing** misses are those taken on core X as a result of a write from core Y to a different part of a cache line needed by X. **Conflict** misses are those taken because a core uses, in rapid succession, too many distinct items of data that happen to fall in the same associativity set. Finally, **capacity** misses are those taken because the working set of the software is larger than the total cache capacity.

Identifying the specific causes of cache misses can be difficult, even with tools such as CPU time profilers or hardware-counter based cache miss profilers. For example, if many different points in a program read a particular variable, and the variable is frequently modified, the miss costs will be distributed widely and thinly over a CPU time profile; no one profile entry will attract the programmer's attention. Capacity misses may have the same lack of focused symptoms, because a cache that is too small for the active data will cause misses throughout the software; even if the programmer realizes that the cache miss rate is higher than it should be, it is rarely clear what data is contributing to the too-large working set. Conflict misses similarly can be difficult to spot and to attribute to specific items of data.

Creating a sharp distinction between misses due to invalidations, capacity misses, and associativity misses is important because the strategies to reduce misses are different in the three cases. Associativity conflicts can usually be fixed by allocating memory over a wider range of associativity sets. False cache line sharing can be fixed by moving the falsely shared data to different cache lines. True sharing can sometimes be reduced by factoring data into multiple pieces that usually need only be touched by a single CPU, or by restructuring software so that only one CPU needs the data. Avoiding capacity misses may require changing the order in which data is processed to increase locality, or imposing admission control on the number of concurrent activities.

This thesis describes a new tool called DProf whose goal is to help programmers understand and eliminate cache misses. DProf addresses two core challenges: identifying the causes of misses, and presenting those causes in a way that helps the programmer eliminate them. DProf uses performance monitoring hardware to gradually accumulate traces of running software's references to memory addresses. These traces allow DProf to categorize all types of cache misses. Associativity misses are identifiable as repeated cycling of the same addresses in a single associativity set. True and false sharing are identifiable by comparing traces from different cores. DProf identifies capacity misses by estimating the working set size from the traces.

Once DProf has categorized cache misses, it identifies the types of data participating in each miss. DProf allows the programmer to navigate four different *views* of the cache misses incurred by the software's data types. The highest-level view is a "data profile": a list of data types, sorted by how many misses they suffer. The "miss classification" view indicates the kinds of misses incurred by each data type. The "working set" view shows which data types were most active, how many objects of each type were active, and which associativity sets those objects used; this information helps diagnose capacity and associativity misses.

Finally, to find instances of true and false sharing, the "data flow" view shows when data moves from one CPU to another.

We have implemented DProf within the Linux kernel, using Linux's kernel allocator to help identify the type of each memory location that misses. The implementation uses the AMD instruction-based sampling (IBS) hardware [10] and $x86$ debug registers [13] to acquire a trace of references.

To evaluate the effectiveness of DProf, we present a number of case studies. Each involves a workload that generates significant misses within the kernel. In each case, existing tools such as OProfile do not usefully identify the cause of the misses. We show that DProf does identify them, and presents the programmer with information that is useful in understanding and eliminating the misses.

The rest of this thesis starts with a comparison of DProf to related work in Section 2. Section 3 presents the views that DProf provides. Section 4 describes how DProf generates those views, and Section 5 details our data collection mechanisms for $x86$ processors. Section 6 uses case studies to show how DProf can be used to find a range of problems related to cache misses in the Linux kernel. Section 7 mentions limitations and directions for future research, and Section 8 concludes.

Chapter 2

Related Work

An overwhelming number of papers have studied the cache performance of applications, and the strong impact of cache access patterns on the performance has been noted for Unix kernels [7] and databases [3] many years ago. Recent work has tried to provide tools to help programmers make more efficient use of caches for specific workloads [17, 6]. DProf, on the other hand, helps programmers *determine* the cause of cache misses that lead to poor system performance, so that they know where to either apply other tools, or make other changes to their code. Tools most similar to DProf can be subdivided into two categories: Section 2.1 describes code profilers and Section 2.2 describes data profilers.

2.1 Code Profilers

Many performance issues, data related included, are dynamic and depend on runtime conditions. Code profilers are a popular class of tools used to gather runtime statistics about an application or the entire system. At their most basic level, code profilers show how long functions in an application have executed during a profiling session. Code profilers rank functions by popularity so that a developer can find potential problems and improve the performance of the application by looking at the top time consumers. Unfortunately, in a world where instructions are cheap but memory accesses are expensive, these tools fail to elucidate a major source of problems: the data. Instruction pointers only indirectly capture how data is used and thus code profilers cannot directly help find problems with an application's data access patterns.

Given a function with high execution time, only some code profilers can detect that cache misses are adding delays to the execution. Even if code profilers can find delays due to cache misses they lack context to make use of the information: code profilers can only aggregate samples by instruction or function. For example, the same data can be accessed by many different functions. If a particular data is habitually suffering cache misses, the misses are distributed among all the functions that access the data. This distribution of samples can water-down the significance of the cache misses. In this situation, the code profiler is not aware that the different instructions do in fact share a common thread: the data. It would be best to aggregate samples by data to bring the cache misses into focus.

Even if instructions with high cache misses do stand out, code profilers offer no help in finding why cache misses are occurring. Given a cache miss, code profilers cannot distinguish if a miss is due to an invalidation or an eviction. Code profilers will not know which line of code was responsible for an invalidation because they do not know when data is shared across CPUs. Code profilers cannot help with capacity misses because they do not know whether the working set of an application exceeds the capacity of the caches.

Nevertheless, code profilers are the most commonly used tools to locate performance problems. Gprof [11] has long been used to help developers find bottlenecks in their code. Gprof aims to highlight the functions that take the most time to execute and provide the context, in the form of call graphs, in which the functions were executed. Gprof's main tenet is the highest gain comes from improving the most executed function. To prevent high overhead gprof finds the most executed functions by periodically sampling the instruction pointer. The output of gprof is very simple and powerful: a list of functions ordered by execution time. A developer knows exactly where to look to find a potential problem and need only to start at the top and work down to improve the performance of an application. Unfortunately, gprof is written to profile sequential programs [5] and cannot differentiate cache miss stalls from other stalls.

Quartz [5] is an application profiler that replicates gprof's simple interface but extends gprof's functionality to parallel applications. Quartz modifies gprof's most executed time metric by biasing the execution time of the least parallel functions. With Quartz, the output is a list of function names ordered by most executed and least parallel code. Quartz works by instrumenting the threading library used by an application. Every use of a synchronization primitive, such as a spinlock or barrier, and every context switch is logged to a per-CPU shared memory buffer. Periodically, a dedicated CPU samples all CPU logs to get a snapshot of the application. From the log, Quartz can figure out if a CPU was idle, busy waiting, or working. Quartz calculates the importance of a function by dividing the function's execution time by the number of busy CPUs at the same time. The more busy CPUs, the more parallel the code and the less time the function accumulates in the final profile. To use Quartz, applications must run on top of an instrumentable threading library. Even if possible, the instrumentation requires recompilation and can have high overhead. Although Quartz can profile parallel applications, it cannot detect cache misses.

More modern tools use hardware features to improve code profiling. OProfile [18] (which was inspired by DCPI [4]) works by counting hardware events and assigning them to instruction pointers. Using hardware counters, profiling can be based on many characteristics of the system: clock cycles, cache misses, memory accesses, branch miss-predictions, etc. The output of a profiling session is very similar to other code profilers: list of functions ranked by importance. OProfile does not track all possible events in the system, it instead uses statistical sampling to capture the most popular instructions and thus achieves low overhead. Because OProfile uses hardware counters which can count cache related events, it can list specific instructions with cache related problems. OProfile cannot, however, aggregate events from instructions that use the same data, find the reason for cache misses, or point to the code responsible for misses.

Both OProfile and DProf use hardware support for sampling performance information. In particular, DProf relies heavily on $x86$ debug registers [13] and IBS [10], which was inspired by ProfileMe [8]. However, DProf differs from most other sampling-based profilers in the way that it aggregates the resulting samples.

2.2 Data Profilers

Neither gprof, Quartz nor OProfile make it easy to see how data is used in the system. The authors of MemSpy [19] acknowledge that data access patterns which do not use the cache effectively have a significant impact on performance. Their system aims to help developers locate poor cache access patterns. MemSpy uses a system simulator to execute an application. The simulator allows MemSpy to interpose on all memory accesses and build a complete map of the cache. MemSpy can account for and explain every single cache miss and using a processor accurate model can approximate memory access latencies. Unfortunately, MemSpy has very high overhead and requires applications to run on top of a simulator. It is also harder to calculate access latencies on today's out-of-order processors.

A number of other cache simulators have been proposed [22], including CProf [16] and valgrind's cachegrind module [20, 21]. Most of these simulators are similar to MemSpy, differing in the amount of programmer annotations or instrumentation necessary to run the tool. The key difference from DProf is that none of these tools report cache misses for a particular data structure, making it easy to overlook cache misses that are spread throughout the application's code.

Intel's performance tuning utility (PTU) [14], a follow-on to VTune [15], uses Intel's PEBS facility to precisely associate samples with instructions. The PEBS hardware is similar in purpose to IBS. Both Intel PEBS and AMD IBS can capture the addresses used by load and store instructions and access latencies for load instructions. DProf can use PEBS on Intel hardware to collect statistics.

Intel PTU does not associate addresses with dynamic memory; only with static memory. Collected samples are attributed to cache lines, and if the lines are a part of static data structures, the name of the data structure is associated with the cache line. This is mainly geared at profiling access to static arrays. In addition, there is no aggregation of samples by data type; only by instruction. Intel PTU uses the addresses of collected load and stores to calculate the working set of the application. The working set, however, is presented in terms of addresses and not data types.

The Intel hardware has a richer set of performance counters than the AMD hardware. False cache line sharing is detected by collecting a combination of hardware counters that count local misses and fetches of cache lines in the modified state from remote caches.

Chapter 3

DProf Views

After a programmer has run software with DProf profiling, DProf offers the programmer the following views:

Data Profile The highest level view consists of a data profile: a list of data type names, sorted by the total number of cache misses that objects of each type suffered. As a convenience, the data profile view also indicates whether objects of each type every "bounce" between cores. This view is most useful when the design and handling of each data type is what causes misses for that type. A data profile view is less useful in other situations; for example, if the cache capacity is too small, all data will suffer misses. Aggregation by type helps draw attention to types that suffer many misses in total spread over many objects. The "data profile view" columns of Tables 6.1, 6.4, and 6.5 show examples of this kind of view.

Miss Classification This view shows what types of misses are most common for each data type. For example, objects of type `skbuff` might incur mostly capacity misses, objects of type `futex` mostly associativity conflict misses, and objects of type `pktstat` both true and false sharing misses. For capacity or conflict misses, the programmer would look next at the working set view; for sharing misses, the data flow view.

Working Set This view summarizes the working set of the profiled software, indicating what data types were most active, how many of each were active at any given time, and what cache associativity sets are used by each data type. This global knowledge of cache contents is important to track down the causes of capacity and conflict misses, which are caused by different data elements evicting each other from the cache. The working set column of Tables 6.1, 6.4, and 6.5 illustrate an example of this kind of output, although the full working set view also includes a distribution of data types by cache associativity sets. Using the working set view, the programmer can determine if the software is using too many distinct data items of a particular type at once. The distribution of data types by associativity set, can also help determine if certain data types are aligned with each other, causing conflict misses.

Data Flow Finally, the data flow view shows the most common sequences of functions that reference particular objects of a given type. The view indicates points at which it is common to see a change in which core is using an object or its cache lines. Figure 6-1 shows an example data flow view.

Programmers can use the data flow view to pin down the exact points at which their software transfers cache lines between CPUs, incurring cache misses due to either false or true sharing. Programmers can also use the data flow view to get a better understanding of how their application processes a particular type of data, as we will illustrate in one of the case studies.

Chapter 4

Generating Views

DProf collects two kinds of data to help it generate views. The first kind is *path traces*. Each path trace records the life history of a particular data object, from allocate to free, in terms of the sequence of instruction locations that read or write the object. This sequence may be from a single thread of execution, but may also involve instructions from multiple cores that are using the same object concurrently. DProf collects path traces for a randomly selected subset of the objects allocated during execution, and combines the traces of objects of the same type if those objects are touched by the same sequence of instructions. For each accessing instruction in a combined trace, DProf records the hit probability of the instruction

Average timestamp	**Program counter**	**CPU change**	**Offsets**	**Cache hit probability**	**Access time**
0	`kalloc()`	no	0–128	—	0
5	`tcp_write()`	no	64–128	100% local L1	3 ns
10	`tcp_xmit()`	no	24–28	100% local L1	3 ns
25	`dev_xmit()`	yes	24–28	100% foreign cache	200 ns
50	`kfree()`	no	0–128	—	0

Table 4.1: A sample path trace for a particular data type and execution path. This path trace is for a network packet structure and the transmit path. The CPU change flag indicates whether that program counter was encountered on the same CPU as the previous one, or on a different CPU. The offset indicates the offset into the data structure that was accessed at this program counter location. The cache hit probabilities indicate the percentage of time that memory access was satisfied using different caches in the system, and the access time indicates the average time spent waiting for that memory reference. An execution path is defined by the sequence of program counter values and CPU change flags, and DProf keeps track of how frequently each execution path is seen for each data type.

in different levels of the cache hierarchy, the average time to access the data, the average time since the object's allocation, and a flag indicating that the instruction executed on a different core than the previous instruction. DProf records how frequently each execution path is observed for a given data type. Table 4.1 shows a path trace for a network packet data structure, on an execution path that transmits the packet.

The second kind of data is called an *address set*, which includes the address and type of every object allocated during execution. DProf uses the address set to map objects to specific associativity sets in the cache. Thus, it is sufficient to store addresses modulo the maximum cache size in this data structure.

DProf is a statistical profiler and assumes a workload that is uniform over time. This assumption is similarly made by other time and hardware-counter based profilers.

Path traces are constructed by combining data generated by hardware-counters and traces of references to data gathered using hardware debug registers. We will describe how the path traces and address sets are collected in the next section, but for now we focus on the problem of generating different views using these data sources.

4.1 Data Profile

Recall that a data profile reflects the miss rates and CPU bouncing information for a given type (for example, see Tables 6.1, 6.4, and 6.5). To construct a data profile for type T, DProf combines all of the path traces for T. The CPU bounce flag in T's data profile will be set if any path trace for T indicates a CPU change. The miss rate for T's data profile is the average miss rate to DRAM or other CPUs' caches encountered on all of the execution paths, according to the path traces, weighted by the frequency with which that path is observed.

4.2 Working Set

The working set view presents two kinds of information. First, an indication of whether some cache associativity sets suffer many more misses than others (suggesting associativity conflicts), and the data types involved in those misses. Second, an estimate of which types are most common in the cache, to help diagnose capacity misses.

DProf runs a simple cache simulation to generate this information. DProf randomly picks address sets (weighted by how common each set is) and simulates the memory accesses indicated by each set. When an object is freed in its path trace, that object's cache lines are removed from the simulated cache.

Based on the cache simulation, DProf counts how many distinct pieces of memory are ever stored in each associativity set. The associativity sets with the highest counts are the most likely to be suffering associativity conflict misses. The histogram can be used by the programmer to find what data types are using highly-contended associativity sets and thereby causing conflict misses. DProf reports which types an associativity set holds and the number of distinct instances of each type in the set.

DProf also counts the number of each data type that is present in the cache, averaged over the simulation. The programmer can use these counts to suggest ways to reduce the working set size to avoid capacity misses. Finding the data types that dominate the working set may not always reveal why there are so many instances of these types. For example, in the Linux kernel each buffered network packet is held in an instance of the `skbuff` type. `skbuffs` can be used in many different ways; for example, for TCP FIN, ACK, or data packets. If the cache is full of `skbuffs`, the programmer needs to know which of the many potential sources is generating them. To solve this problem, DProf's working set view reports not only the most common data types found in the cache, but also the execution paths that those data types take.

4.3 Miss Classification

For the miss classification view, DProf uses path traces to classify cache misses into three categories: invalidations (including both true sharing and false sharing), conflict misses, and capacity misses. Although compulsory misses are often treated as a separate class of cache miss, in practice all memory on a real system has been accessed by a CPU at some point in the past, so there are almost no compulsory misses. (Device DMA could cause a compulsory miss, but DMA-to-cache is common and would largely avoid such misses.) Thus, DProf assumes there are no compulsory misses.

Invalidations Invalidations occur when one core has data in its cache and a second core writes the data; the processor's cache coherence protocol removes the data from the first core's cache, and that core's next access to the data will miss. DProf uses path traces to identify misses due to invalidations, and thus to identify instructions that cause invalidations. For each miss in each path trace, DProf searches backwards in the trace for a write to the same cache line from a different CPU. If there is such a write, then the miss is due to invalidation.

False sharing occurs when multiple objects share a cache line. To help catch such problems, DProf coalesces all objects that share a cache line into the same path trace, at which point false sharing becomes apparent.

Conflict Misses An N-way set associative cache is composed of multiple associativity sets. When a cache line is added to the cache, the address of the cache line is used to deterministically calculate which set the cache line maps to. Each set can hold up to N cache lines simultaneously. Conflict misses take place when the software frequently accesses more than N different cache lines that map to the same associativity set. To detect conflict misses, DProf must first determine what associativity sets a particular data type falls into, and then determine whether those associativity sets are used to store significantly more data than other sets.

To determine the associativity sets used by type T, DProf adds the addresses from T's address set to the offsets accessed in T's path traces, and maps the resulting absolute addresses into associativity sets. To determine whether a particular associativity set is suffering from conflict misses, DProf computes the histogram of associativity sets, described earlier in the working set view. DProf then checks whether that associativity set is assigned more cache lines than it can hold, and if so, whether the number of cache lines assigned to it is much higher than the average number of cache lines assigned to other associativity sets. (In our prototype, we check if the number of cache lines is a factor of 2 more than average.)

If both of those conditions hold, DProf marks that associativity set as suffering from conflict misses.

Capacity Misses Capacity misses occur when the total amount of data actively used by the software (the working set) is greater than the size of the cache. DProf helps the programmer understand capacity misses by estimating the primary contents of the working set.

There is overlap between conflict and capacity misses, since at a detailed level capacity misses are caused by associativity conflicts. DProf distinguishes between the two cases heuristically: if only a few associativity sets have lots of conflicts, DProf attributes the misses to conflicts; if most associativity sets have about the same number of conflicts, DProf indicates that the problem is capacity.

4.4 Data Flow

DProf constructs the data flow view for a particular type T by combining the execution paths seen in T's path traces into a single graph. All path traces start from an allocation routine like `kalloc()`, and end in a deallocation routine like `kfree()`. As a result, DProf can always construct a data flow graph from `kalloc()` to `kfree()`. When multiple execution paths share a prefix or suffix, DProf can merge the common program counter values into a shared sequence of nodes in the graph.

Chapter 5

Collecting Path Traces and Address Sets

As Section 4 outlined, DProf relies on two data structures to generate its views: the path traces and the address set. DProf constructs an address set by instrumenting the allocator to record allocation and deallocation of all objects, along with their type information.

DProf pieces together path traces gradually, by sampling limited amounts of raw data on many executions of the same workload. During each execution DProf uses low-overhead CPU performance monitoring features to collect a little more raw data. Finally, DProf combines the raw data into a set of path traces, one per data type / path combination. Because DProf samples and combines data from many executions, its path traces are statistical estimates.

DProf collects two kinds of raw data: *access samples* and *object access histories*. Each access sample records information for an instruction execution randomly chosen by the hardware, and includes the instruction address, data memory address, whether the memory access hit in the cache, and the memory access latency (for misses). An object access history is a complete trace of all instructions that read or wrote a particular data object, from when it was allocated to when it was freed. These two types of raw data are dictated by the performance monitoring hardware provided by the CPUs we use. In particular, the hardware that collects object access histories does not record cache miss information. Combining access samples with object access histories allows construction of path traces.

DProf performs two additional steps in generating path traces. First, it finds the type of the data referenced by each access sample, to help it decide which object access histories

Field	Description
`type`	The data type containing this data.
`offset`	This data's offset within the data type.
`ip`	Instruction address responsible for the access.
`cpu`	The CPU that executed the instruction.
`stats`	Memory access latency and cache information.

Table 5.1: An access sample that stores information about a memory access.

each access sample might be relevant to. Second, it aggregates object access histories that refer to the same data type and involve the same path of instructions.

Currently DProf stores all raw samples in RAM while profiling. Techniques from DCPI [4] can be used to transfer samples to disk while profiling. The following subsections detail how DProf collects and combines the raw data.

5.1 Access Samples

To gather information about cache misses, DProf samples data accesses. Each sampled data access is saved as an "access sample" shown in Table 5.1. The output of a data access profiling session is a collection of access samples that captures the application's popular data accesses and how many of the accesses missed in the cache.

DProf uses Instruction Based Sampling (IBS) [10] provided by AMD processors to implement statistical sampling. IBS is a hardware feature that allows detailed CPU execution sampling. IBS works by randomly tagging an instruction that is about to enter the CPU's pipeline. As a tagged instruction moves through the pipeline, built-in hardware counters keep track of major events like cache misses, branch miss-predictions, and memory access latencies. When the tagged instruction retires, the CPU issues an interrupt alerting the operating system that an IBS sample is ready. In addition, IBS also reports the instruction address for each sample and the physical and virtual memory addresses for instructions that access memory. DProf is a statistical sampler because IBS can only track one instruction at a time in the pipeline. In this sense IBS is limiting; however, DProf uses IBS because it reports a rich set of statistics that otherwise would not be possible to collect.

DProf uses the IBS hardware to generate access samples. After initializing the IBS unit, DProf receives interrupts on each new IBS sample. The DProf interrupt handler creates a new access sample and fills in `stats` with the IBS data. Specifically, DProf records whether the access resulted in a cache miss, where in the cache hierarchy the data come from, and how long it took to load the data. The instruction address is saved in `ip` and `cpu` is filled in with the current CPU number. Finally, DProf resolves the memory address provided by IBS into the `type` of the data structure containing the address, and the `offset` in that data structure corresponding to the address, as we will describe next.

5.2 Address to Type Resolution

AMD's IBS mechanism captures the memory address of the accessed data. To construct an access sample, however, DProf needs to compute the type and offset that corresponds to the memory address. The type information is necessary to group together accesses to objects of the same type, even if they are allocated at different addresses, and to differentiate accesses to the same address that may be recycled during the program's execution.

DProf assumes C-style data types, whose objects are contiguous in memory, and whose fields are located at well-known offsets from the top-level object's base address. DProf implements a memory type resolver whose job is to generate a type and offset for any memory address at runtime. For a given address, the resolver finds the type of object that the address belongs to, and the base address of that object. Subtracting the address from the base address gives the offset into the type, which can be used to infer the field.

The method for finding the type and base address of an address depends on whether the address refers to dynamically-allocated or statically-allocated memory. For statically-allocated memory, the data type and the object base address can be found by looking at the debug information embedded in the executable.

For dynamically-allocated memory, DProf modifies the allocator to keep track of the type of all outstanding allocations. The Linux kernel's allocator already has this information, since a separate memory pool is often used for each type of object. As a result, DProf can ask the allocator for the pool's type, and the base address of the allocation that includes a given

Field	Description
`offset`	Offset within data type that's being accessed.
`ip`	Instruction address responsible for the access.
`cpu`	The CPU that executed the instruction.
`time`	Time of access, from object allocation.

Table 5.2: An element from an object access history for a given type, used to record a single memory access to an offset within an object of that type.

address. We are exploring off-line assembly typing techniques for associating types with dynamically-allocated memory from shared memory pools, and for allocators outside of the Linux kernel. For now, our approach has been to manually annotate with type information the few allocations that come from a generic pool.

5.3 Object Access Histories

Object access history profiling works by recording all instruction pointers that access a given object during that object's lifetime. An object access history is a collection of elements shown in Table 5.2.

DProf uses debug registers provided by modern AMD and Intel CPUs to track loads and stores to a given memory location. Debug registers are a hardware feature that allow the OS to register a particular memory location with the CPU, and have the CPU interrupt the OS every time a load or store is issued to that memory location.

Current hardware provides a limited number of debug registers, each covering up to eight bytes of contiguous memory at a time. As a result, DProf must use debug registers to monitor a small part of each data structure at a time, and to piece together the resulting offset access histories into a complete object access history.

DProf monitors one object at a time. When a free debug register becomes available, DProf decides what type of object it would like to monitor, focusing on the most popular objects found in the access samples. DProf also picks an offset within that data type to monitor, based on what offsets have not been covered yet. DProf then cooperates with the kernel memory allocator to wait until an object of that type is allocated. When an allocation happens, DProf configures the debug registers on every CPU to trace the given offset within the newly-allocated memory region, until the object is eventually freed, and the debug register is released.

After a traced object is allocated, the CPUs fire an interrupt on each load and store to a specific offset within that object. The DProf interrupt handler creates a record of the access on each interrupt, as shown in Table 5.2. The `offset` field is known at allocation time, the `ip` and `cpu` fields are known at interrupt-time, and `time` field is computed using the `RDTSC` timestamp counter, relative to the object's initial allocation time.

In order to build up an object access history for all offsets in a data type, DProf must determine how accesses to different offsets should be interleaved with one another. To do so, DProf performs pairwise sampling using pairs of debug registers, configured to track two different offsets within the same object. DProf samples all possible pairs of offsets within an object, and then combines the results into a single access history by matching up common access patterns to the same offset. While this process is not perfect, access patterns for most data types are sufficiently repetitive to allow DProf to perform this merging.

5.4 Path Trace Generation

Once the access samples and object access histories have been collected, DProf combines the two data sets to create path traces for each data type. First, DProf aggregates all access samples that have the same `type`, `offset`, and `ip` values, to compute the average cost of accessing memory of a particular type by a given instruction. Then, DProf augments object access histories with data from the access samples, by adding the `stats` data from the access sample to object access history records with the same `type`, `offset`, and `ip` values. Finally, DProf combines all augmented object access histories that have the same execution path (same sequence of `ip` values and equivalent sequence of `cpu` values) by aggregating their `time` and `stats` values. All of the combined histories for a given type are that type's *path trace*.

Chapter 6

Evaluation

This section evaluates DProf by using it to find cache performance problems in the Linux kernel for two workloads, Apache and memcached. The hardware involved is a 16-core AMD machine with an Intel 10GB Ethernet card (IXGBE) [9]. Sixteen other machines generate load via a 24-port 1GB switch with a 10GB Ethernet interface.

The IXGBE has 16 TX and RX queues. Each queue can be set to interrupt one specific core. The card hashes incoming packets to choose the queue. We configured the card to ensure that each load-generation host's packets went to a different core, in order to reduce contention over a number of kernel locks and data structures. We also modified the kernel to avoid using global data structure locks in subsystems like the directory entry cache and the SLAB memory allocator.

In this section, we present two case studies of how DProf helps find cache access problems. We also compare how precisely DProf and two existing popular tools, lock_stat and OProfile, directed the programmer's attention to the underlying causes of the problems. Lock_stat reports, for all Linux kernel locks, how long each lock is held, the wait time to acquire the lock, and the functions that acquire and release the lock. OProfile [18] is an execution profiler, reporting the cost of each function or line of code.

Type name	Description	Working Set View	Data Profile View	
		Size	% of all L1 misses	Bounce
size-1024	packet payload	14.6MB	45.40%	yes
slab	SLAB bookkeeping structure	2.55MB	10.48%	yes
array_cache	SLAB per-core bookkeeping structure	128B	9.51%	yes
net_device	network device structure	128B	6.03%	yes
udp_sock	UDP socket structure	1024B	5.24%	yes
skbuff	packet bookkeeping structure	20.55MB	5.20%	yes
Total		37.7MB	81.86%	—

Table 6.1: Working set and data profile views for the top data types in memcached as reported by DProf.

6.1 Case Study: True Sharing

Memcached [2] is an in-memory key-value store often used to speed up web applications. A distributed key-value store can be created by running multiple instances of memcached and having clients deterministically distribute keys among all available servers.

For this case study, the test machine ran 16 instances of memcached. Each instance used a different UDP port and was pinned to one core. Separate memcached processes were used to avoid scalability bottlenecks with threaded memcached. Each load generating machine ran a UDP memcached client and queried the memcached instance running on the same core that serviced the packet of that load generating machine. Each UDP client repeatedly asked for one non-existent key.

This configuration aimed to isolate all data accesses to one core and eliminate cross core sharing of data that results in both contention for locks and cache misses. Even though we configured the experiment to reduce cross core sharing, we were still not getting linear speed up running memcached.

6.1.1 Profiling with DProf

Table 6.1 shows the data profile generated by DProf. A few kernel objects had a high concentration of cache misses. In addition, the same objects were bouncing between cores. With the highest proportion of the cache misses, the `size-1024` objects are a

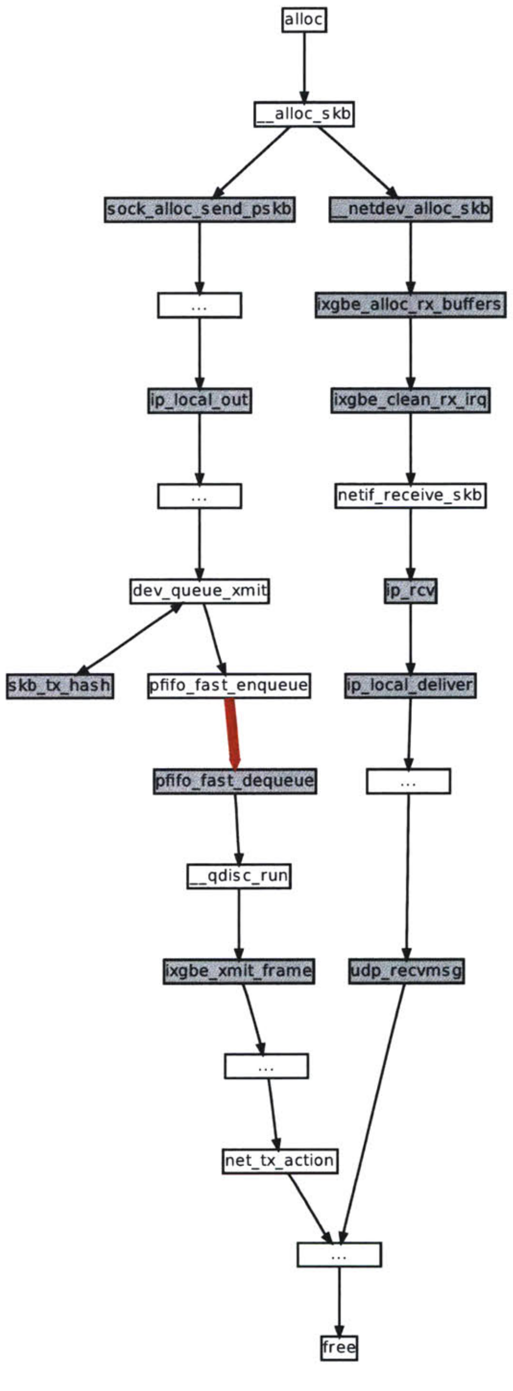

Figure 6-1: Partial data flow view for `skbuff` objects in memcached as reported by DProf. Bold lines indicate a transition from one core to another. Darker boxes represent functions with high cache access latencies.

good place to start the analysis. `size-1024` objects hold packet payload. Diving into the `size-1024` data flow view, DProf shows a large portion of objects jumping from one core to another between a call to `dev_queue_xmit` and `dev_hard_start_xmit`. This means that individual packets are handled by multiple cores during transmit processing, rather than by one as expected. The `size-1024` objects are only the packet payloads; per-packet `skbuff` objects are used to store bookkeeping information about each packet. Since `skbuffs` are on the list and are also bouncing, they are worth looking at next.

Figure 6-1 shows a snippet of the data flow view generated by DProf. This data flow graph indicates that `skbuffs` on the transmit path jump from one core to another between a call to `pfifo_fast_enqueue` and `pfifo_fast_dequeue`. Both of these functions are part of the `Qdisc` Linux subsystem that is used to schedule packet transmission. Packets are placed on the head of the queue and are taken off the queue when the card is ready to transmit them. As described earlier, the IXGBE driver was configured to create 16 hardware transmit queues. In principle, each core should be able to transmit a packet without contending for locks by placing the packet onto a "local" queue dedicated to its use. Since `skbuffs` jump to a different core at this point, this is apparently not happening.

Now that we have an idea of what the problem is, we need to find why packets are not placed on the local queue. The data flow graph limits the scope of the search: we only need to look at functions above `pfifo_fast_enqueue` to find why packets are not placed on to the local queue. Fortunately, we do not need to look far for the problem. Looking at the `skb_tx_hash`, it is easy to spot that the transmit queue is chosen by hashing the content of the `skbuff`.

The problem is that the IXGBE driver does not provide its own custom queue selection function that overrides the suboptimal default. In the memcached benchmark it is best to setup the system to choose the local queue over balancing transmit load by hashing to a remote queue. Implementing a local queue selection function increased performance by 57% and eliminated all lock contention.

Lock Name	Wait Time	Overhead	Functions
epoll lock	0.6594 sec	2.20%	`sys_epoll_wait`, `ep_scan_ready_list`, `ep_poll_callback`
wait queue	0.5658 sec	1.89%	`__wake_up_sync_key`
Qdisc lock	1.2134 sec	4.04%	`dev_queue_xmit`, `__qdisc_run`
SLAB cache lock	0.0477 sec	0.16%	`cache_alloc_refill`, `__drain_alien_cache`

Table 6.2: Lock statistics reported by lock_stats during a 30 second run of memcached.

6.1.2 Analysis using Lock_stat

We wanted to see how easy it would be to find the same problem using lock_stat. Lock_stat found four contended kernel locks shown in Table 6.2. Contention for locks implies that the memory protected by the lock is accessed on multiple CPUs, and might be a bottleneck. To try to use lock_stat's results to understand why memcached wasn't scaling, we looked at each lock it reported.

The first reported lock protects part of the event poll system call interface. In the memcached benchmark it might be safe to assume that socket descriptors generate most events. The event poll subsystem uses wait queues internally, so the "epoll" and "wait queue" lock uses might be related. Without call stacks, it is hard to tell how the two locks relate, or how other parts of the kernel might be using events or wait queues.

Contention over the SLAB cache lock indicates that data is allocated on one CPU and deallocated on another. `cache_alloc_refill` is called to refill the exhausted per-core local cache of free objects. The call to `__drain_alien_cache` shows that objects are frequently freed on a core other than the original allocating core, requiring the object to be placed back on the original core's free lists. It is not clear from the lock_stat output what type of objects are being allocate and deallocated.

The `Qdisc` lock contention indicates that multiple cores are touching the same packet transmit queue. The `dev_queue_xmit` function places packets onto a queue and the `__qdisc_run` function takes them off. Studying the `dev_queue_xmit` function, we could make the same conclusion as we did with DProf. The fact that this is possible is something of a lucky coincidence for lock_stat, since there is no real reason why the choice of queue and the mechanics of placing the packet on that queue had to be in the same function. Often such pairs of actions are not in the same place in the code, and lock_stat often doesn't identify the point in the code that effectively decides to share data with another core. DProf's data flow views identify these problem more explicitly, limiting the functions that need to be examined to the ones that used the data just before the data switched to a different core.

Without knowledge of the select queue policy, a possible solution would be to modify the `Qdisc` subsystem to use lock-less queues. Removing the locks would increase performance, but locks are only one part of the problem; installing a local queue selection policy increased performance by 57%. Data cache contention is still a problem even if locks are eliminated. Once the locks are eliminated (or if the queues where lock-less in the first place), lock_stat would be less useful in analyzing this problem.

6.1.3 Profiling with OProfile

Table 6.3 shows the output of OProfile for the memcached workload. OProfile reports over 29 functions with more than 1% of the samples. It is hard to tell whether the functions actually are causing any problems. The memcached benchmark is forcing the kernel to allocate, transmit, and deallocate millions of packets per second; it is reasonable for `kfree`, `ixgbe_clean_rx_irq`, and `__alloc_skb` to be at the top of the list. Before getting to the interesting `dev_queue_xmit` function, the programmer needs to figure out why the first 6 functions are popular. Even when inspecting `dev_queue_xmit`, the programmer would likely focus on the locking problem, just as in the lock_stat analysis.

The OProfile output mentions 6 functions that deal with packet transmission. Another 14 are generic functions that may manipulate packets. Even if the programmer realizes that the problem is packets placed on a remote queue, up to 20 functions would need to be studied to understand why this is happening.

OProfile does generate call graphs that present information similar to data flow views. For the purposes of this analysis, however, call graphs lack the continuity of DProf data flows when objects are placed in data structures and later retrieved, as with queues.

% CLK	% L2 Misses	Functions
4.4	3.2	kfree
3.7	5.1	ixgbe_clean_rx_irq
3.5	2.9	__alloc_skb
3.2	2.7	ixgbe_xmit_frame
3.0	2.0	kmem_cache_free
2.6	3.4	udp_recvmsg
2.4	2.6	dev_queue_xmit
2.3	2.2	ixgbe_clean_tx_irq
2.1	2.8	skb_put
2.0	1.2	ep_poll_callback
2.0	2.5	copy_user_generic_string
1.9	1.6	__kfree_skb
1.9	3.0	local_bh_enable
1.8	0.39	skb_tx_hash
1.7	0.79	sock_def_write_space
1.6	2.0	ip_rcv
1.5	1.7	lock_sock_nested
1.4	0.60	eth_type_trans
1.4	1.4	getnstimeofday
1.3	0.76	dev_kfree_skb_irq
1.3	0.93	__qdisc_run
1.2	1.6	skb_copy_datagram_iovec
1.2	0.90	__wake_up_sync_key
1.2	0.86	skb_dma_map
1.2	1.4	kmem_cache_alloc_node
1.2	2.5	__phys_addr
1.1	1.4	event_handler
1.0	0.52	ixgbe_set_itr_msix
1.0	1.1	udp_sendmsg

Table 6.3: Top functions by percent of clock cycles and L2 misses for memcached as reported by OProfile.

6.2 Case Study: Working Set

For this case study we configure the test machine to run the Apache [1] web server. Just like in the memcached setup, we run 16 different Apache servers on different ports, each pinned to a different core. Each Apache server is configured to serve a single 1024B static file out of memory pre-cached by the `MMapFile` directive. The 16 load generating machines repeatedly open a TCP connection, request the file once, and close the connection.

Type name	**Description**	**Working Set View**	**Data Profile View**	
		Size	% of all L1 misses	Bounce
`tcp_sock`	TCP socket structure	1.11MB	11.00%	no
`task_struct`	task structure	1.19MB	21.37%	no
`net_device`	network device structure	128B	3.40%	yes
`size-1024`	packet payload	4.23MB	5.19%	no
`skbuff`	packet bookkeeping structure	4.27MB	3.28%	no
	Total	10.80MB	44.24%	—

Table 6.4: Working set and data profile views for the top data types in Apache at peak performance as reported by DProf.

Type name	**Description**	**Working Set View**	**Data Profile View**	
		Size	% of all L1 misses	Bounce
`tcp_sock`	TCP socket structure	11.56MB	21.47%	no
`task_struct`	task structure	1.29MB	10.69%	no
`net_device`	network device structure	128B	12.03%	yes
`size-1024`	packet payload	6.25MB	4.12%	no
`skbuff`	packet bookkeeping structure	7.22MB	3.74%	no
	Total	26.32MB	52.05%	—

Table 6.5: Working set and data profile views for the top data types in Apache at drop off as reported by DProf.

Measuring the performance of Apache we ran into a situation where the number of requests per second the machine could handle dropped off past a certain point. The peak performance was achieved with some percent of all cores idle. Adding any more load maxed out all cores and decreased the number of request per second.

6.2.1 Profiling with DProf

Using DProf we captured two runs of Apache: one at the peak and one when performance dropped off. Looking at the results shown in Table 6.4 and Table 6.5, it was obvious that something happened to the working set size of `tcp_sock` objects. The data flow view also indicated that the time from allocation to deallocation of `tcp_sock` objects increased

Lock Name	Wait Time	Overhead	Functions
futex lock	1.98	6.6%	`do_futex`, `futex_wait`, `futex_wake`

Table 6.6: Lock statistics acquired by lock_stats during a 30 second run of Apache.

significantly from the peak case to the drop off case. Here, we used DProf to perform differential analysis to figure out what went wrong between two different runs.

We realized that we miss-configured the server. Each instance of Apache allowed many TCP requests to be backlogged on its accept queue. The load generating machines eagerly filled this queue with new requests. In the peak performance case, the time it took from when a request was received by the kernel to the time Apache accepted the new connection was short because Apache kept up with the load and the queue remained shallow. When Apache accepted a new connection, the `tcp_sock` was in a cache closer to the core. In the drop off case the queue filled to its limit and by the time Apache accepted a connection, the `tcp_sock` cache lines had already been flushed from the caches closest to the core. The average cycle miss latency for a `tcp_sock` cache lines was 50 cycles in the peak case and 150 cycles in the drop off case.

We implemented admission control by limiting the size of the queues to cut down on the number of in flight TCP connection requests. This change improved performance by 16% when the server underwent the same request rate stress as the drop off point.

6.2.2 Analysis with Lock_stat

The lock_stat analysis results are shown in Table 6.6. The results show that the Linux kernel fast user mutex (futex) subsystem is taking up execution time acquiring locks. This is indeed the case because Apache threads communicate with each other using queues that implement thread wake up by using futexes. This analysis does not reveal anything about the problem.

6.2.3 Profiling with OProfile

OProfile tracks events like context switches and thus does more work than DProf when collecting samples regardless of the sampling rate. We were not able to collect reliable data

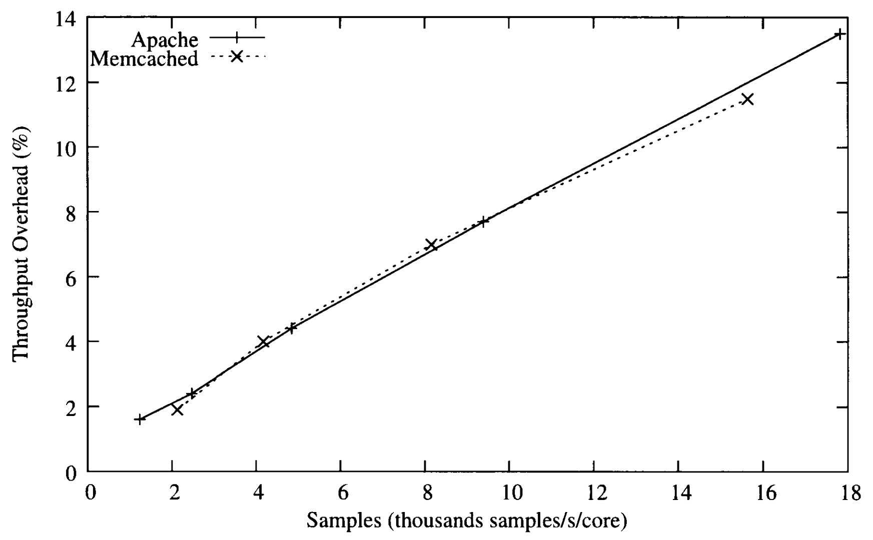

Figure 6-2: DProf overhead for different IBS sampling rates. The overhead is measured in percent connection throughput reduction for the Apache and memcached applications.

with OProfile because any major perturbation to the test machine caused the machine to go from the peak state straight into the drop off state.

6.3 Access Sample Overhead

The overhead of data access profiling comes from taking an interrupt to save the access sample. The overhead is proportional to the IBS sampling rate; Figure 6-2 shows the overhead of profiling for different IBS sampling rates for the Apache and memcached applications. The sampling rate is chosen based on the overhead tolerance. With lower sampling rates it is critical to sample long enough to capture enough access samples of data types in interest.

The cost of an IBS interrupt is about 2,000 cycles on our test machine. Half of the cycles are spent reading IBS data out of a core's IBS registers, the rest is spent entering and exiting the interrupt and resolving the data's address to its type.

Just like the CPU cycle overhead, the memory overhead of data access profiling is proportional to the number of access samples collected. Each access sample is 88 bytes.

Benchmark	Data Type	Data Type Size (bytes)	Histories	Histories Sets	Collection Time (s)	Overhead (%)
memcached	size-1024	1024	8128	32	170	1.3
	skbuff	256	5120	80	95	0.8
Apache	size-1024	1024	20320	80	34	2.9
	skbuff	256	2048	32	24	1.6
	skbuff_fclone	512	10240	80	2.5	16
	tcp_sock	1600	32000	80	32	4.9

Table 6.7: Object access history collection times and overhead for different data types and applications.

Benchmark	Data Type	Elements per History	Histories per Second	Elements per Second
memcached	size-1024	0.3	53	120
	skbuff	4.2	56	350
Apache	size-1024	0.5	660	1660
	skbuff	4.8	110	770
	skbuff_fclone	4.0	4600	27500
	tcp_sock	8.3	1030	10600

Table 6.8: Average object access history collection rates for different data types and applications.

6.4 Object Access History Overhead

The overhead of capturing object access histories depends on the number of debug register interrupts triggered per second and the number of histories collected per second. For the Apache and memcached applications Table 6.7 shows the range of overheads when profiling different data types. A history set is a collection of object access histories that cover every

Data Type	Interrupts	Memory	Communication
size-1024	20%	10%	70%
skbuff	60%	10 %	30%
skbuff_fclone	5%	5%	90%
tcp_sock	20%	5%	75%

Table 6.9: Object access history overhead breakdown for different data types used by Apache. The overhead is composed of the cost to take a debug register *interrupt*, the cost to communicate with the *memory* subsystem to allocate an object for profiling, and the cost to *communicate* with all cores to setup debug registers.

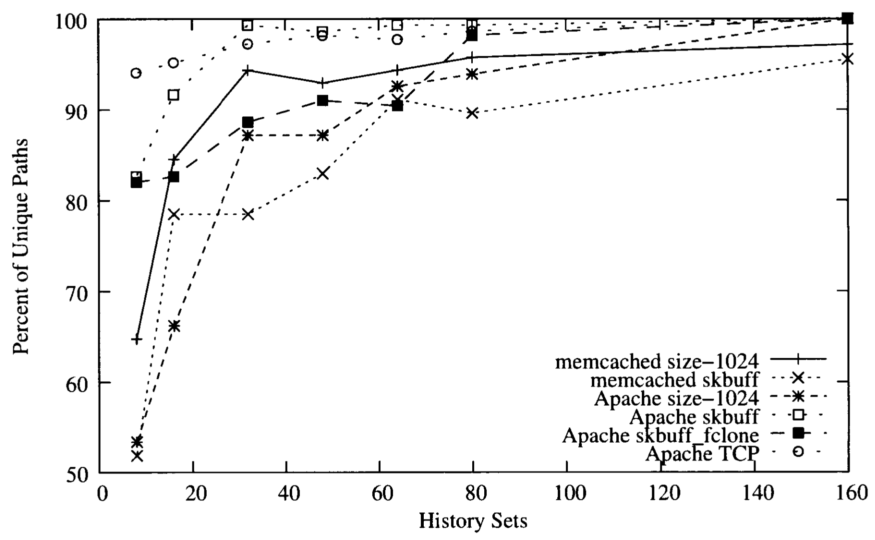

Figure 6-3: Percent of unique paths captured as a function of history sets collected. For all data types, the maximum number of unique paths is based on a profile with 720 history set. Only results for profiles with less than 160 sets are shown.

offset in a data type. For example, a `skbuff` is 256 bytes long and its history set is composed of 64 histories with debug register configured to monitor length of 4 bytes.

Table 6.9 shows a breakdown of the profiling overhead. The first component is the cost to take an interrupt and save an element. This interrupt is triggered every time the profiled object is accessed and costs the test machine 1,000 cycles.

There are two setup overheads: the cost to reserve an object for profiling with the memory subsystem and the cost to setup debug registers on all cores. At high histories per second rates, the dominating factor is the debug registers setup overhead. The core responsible for setting up debug registers incurs a cost of 130,000 cycles. The high cost is due to interrupts sent to all other cores to notify them to set their debug registers.

In addition to cycle overhead, history collection incurs a memory overhead of 32 bytes per element in an object access history.

The time to collect all histories depends on the size and lifetime of an object. Table 6.7 shows collection times for a number of data types. Table 6.8 shows the rates at which histories are collected. Since DProf profiles only a few bytes of an object at a time, the

Benchmark	Data Type	Size (bytes)	Histories/Sets	Time (s)	Overhead (%)
memcached	size-1024	1024	32132/1	400	0.9
	skbuff	256	2017/1	26	1.0
Apache	size-1024	1024	32132/1	50	4.8
	skbuff	256	2017/1	18	1.7
	skbuff_fclone	512	8129/1	2.3	18
	tcp_sock	1600	79801/1	81	5.5

Table 6.10: Object access history collection times and overhead using pair sampling for different data types and applications.

bigger the object the more runs are needed. Also because DProf can profile only a few objects at a time, the longer an object remains in use, the longer it takes to capture its history.

The time to setup an object to be monitored also adds to the profiling time. It costs about 220,000 cycles to setup an object for profiling. Most of this cost comes from notifying all cores to setup their debug registers.

An object can take multiple paths; to record all of an object's paths, DProf needs to profile multiple times until all paths are captured. DProf is concerned with capturing the most often take paths. Paths that are not taken often usually do not contribute to the performance of an application. The question is how many times should an object be profiled to collect all relevant paths? There is no specific answer because the number depends on individual data types. We have found that for all the data types we studied, 30 to 100 history sets where sufficient to collect all relevant paths

To verify this, we collected profiles with a large number of history sets (720 sets per data type) for a couple different data types used by the Apache and memcached applications. By collecting a profile with a large number of history sets we hope to capture all unique and relevant paths for a particular type. We then collected profiles with a decreasing number of history sets and counted the number of unique paths found. Figure 6-3 shows the result of this experiment. As the number of histories collected decreases, the percent of all unique paths captured decreases as well. In general 30 to 100 history sets are sufficient to capture most unique paths.

To build the data flow view graph, DProf needs to monitor multiple bytes of an object per run. This is needed to create an order on accesses to members of an object on each path.

To get a complete picture, DProf profiles every pair of bytes in an object. Pairwise data collection takes longer because many more histories need to be collected to profile an object just once; the increase is quadratic. Table 6.10 shows times for pairwise profiling.

To reduce the time to collect pairwise histories, DProf does not profile all members of a data type. Instead, DProf analyzes the access samples to find the most used members. The programmer can tune which members are in this set. DProf profiles just the bytes that cover the chosen members.

Chapter 7

Discussion

DProf's main challenge is in collecting data profiling information, and it is limited by the available hardware mechanisms—mechanisms provided by AMD and Intel in their latest *x*86 processors. First, DProf estimates working set sizes based on allocation, memory access, and deallocation events. Having hardware support for examining the contents of CPU caches would greatly simplify this task, and improve its precision. DProf is also limited by having access to only four debug registers for tracing memory accesses through application code. As a result, computing object access histories requires pairwise tracing of all offset pairs in a data structure. Collecting precise information in this manner is difficult, and having a variable-size debug register would greatly help DProf.

Chapter 8

Summary

Cache misses are a significant factor in application performance, and are becoming increasingly more important on multicore systems. However, current performance profiling tools focus on attributing execution time to specific code locations, which can mask expensive cache misses that may occur throughout the code. This thesis presented DProf, a data-oriented profiler that attributes cache misses to specific data types. Programmers can use DProf's data-oriented views to locate and fix different causes of cache misses in their applications, which may be difficult to locate using CPU profilers. We have used DProf to analyze cache misses encountered in the Linux kernel, and have found and fixed a number of suboptimal memory access patterns. Our fixed Linux kernel achieves 16–57% throughput improvement running a range of memcached and Apache workloads.

Bibliography

[1] Apache HTTP Server, January 2010. `http://httpd.apache.org/`.

[2] Memcached, January 2010. `http://memcached.org/`.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, September 1999.

[4] 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? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[5] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.

[6] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[7] J. Chapin, A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Ottawa, Ontario, May 1995.

[8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[9] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[10] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors, November 2007. `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`.

[11] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, MA, 1982.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[13] Intel. *Intel 64 and IA-32 Architectures Developer's Manual*, November 2008.

[14] Intel. PTU, January 2010. `http://software.intel.com/en-us/articles/intel-performance-tuning-utility/`.

[15] Intel. VTune, January 2010. `http://software.intel.com/en-us/intel-vtune/`.

[16] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27:15–26, 1994.

[17] R. Lee, X. Ding, F. Cheng, Q. Lu, and X. Zhang. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Proceedings of the 35th International Conference on Very Large Data Bases*, Lyon, France, August 2009.

[18] J. Levon et al. Oprofile, January 2010. `http://oprofile.sourceforge.net/`.

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

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, CA, June 2007.

[21] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. Invited tutorial, IEEE International Symposium on Workload Characterization, October 2006. `http://valgrind.org/docs/iiswc2006.pdf`.

[22] J. Tao and W. Karl. Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In *Proceedings of the 1st International Conference on Performance Evaluation Methodologies and Tools*, Pisa, Italy, October 2006.