SLEEPY: Sampling User-Level Execution via EIPs Profiled Continuously

Michelle Joy Moravan   Margaret Ann Richey

May 10, 2005

Abstract

Understanding where the system is spending its time is necessary for debugging and optimizing applications. As such, we present an analysis tool with the intent to continuously take measurements on every timer interrupt during a sampling period. The end result is a collection of data samples that have been aggregated to display what instruction pointers are seen most often. These aggregates can be viewed on an instruction level, an instance of an application, a subset of the running applications, or on the system as a whole. Based on the benchmarks, the majority of the results are verifiable and the sampling itself is extremely efficient with minimal overhead (0.01 - 0.9 percent). The overhead of the system utilizing the worst case backup scheme is three to six percent.

1 Introduction

Understanding the behavior of applications is essential to obtaining optimal performance. However, this is often hard to determine. Is poor performance the result of the poor design of an application or is there a problem with the interaction between multiple applications?

SLEEPY endeavors to answer these questions through the use of continuous profiling. Our implementation provides an accurate and efficient way of determining where processes spend their time. By taking samples of the instruction pointer (EIP) of the currently running process on every timer interrupt, an overall system view can be obtained. The number of times each EIP is sampled is proportional to the time the CPU spends executing that instruction. Thus, after a sampling period has taken place, a front end takes the samples and aggregates the results to display the percentage each EIP occurred.

In addition, SLEEPY has the ability to aggregate on different granularities. This is especially important when zeroing in on certain applications, functions, or even instructions. The additional view of the system-wide performance gives the effect of multiple applications interacting with each other versus simulating the performance of a system without taking into account the timesharing nature of modern operating systems. Both views are important in determining the bottlenecks within system and application performance.

Profiling can also determine memory requirements of applications. The amount of memory used and the areas accessed most often can be determined by looking at the instruction pointer. Again, this can impact the design of data structures or access patterns of data during runtime.

Therefore, the use of such a profiling system is manifold. This profiling enables application and systems engineers to find bottlenecks within their work. SLEEPY can be used as a debugging and optimizing analysis tool.

It should also be mentioned that SLEEPY is focusing on user-level events. The Linux kernel already provides a profiler for kernel operations.

Our main contributions are the methods used to enable sampling. In this paper, we explore techniques for efficiently and accurately measuring statistics on every timer interrupt. The rest of the paper is organized as follows. Section 2 gives a brief outline of the infrastructure for building and testing the tool and describes the methodology used to build (name here). In Section 3, we report our results. We present future directions to extend the scope of the tool in Sections 4 and 7. The related work is discussed in Section 5 and we conclude in 6.

2 Implementation

We limit the scope of this work to continuously profiling the x86 architecture in a Linux environment. Toward this end, we extended the 2.4.21 kernel of Red Hat 7.2. As we proceed, bear in mind that the final output of the kernel portion of our software consists of a stream of pairs, each of which indicates the application that was running and which EIP it was about to execute when our code recorded that sample. In the subsequent subsections, we will describe the usage, control flow, and data structures of our implementation.

2.1 Usage

Despite the original goal of continuous profiling, we allow the user to enable and disable sampling at will. This offers several benefits. Primarily, this interface incurs sampling overheads only when the user actually wishes to obtain profiling information. It also expedited our performance experiments by allowing us to use the same kernel to benchmark execution times with and without sampling. This eliminated the recompilations and reboots that would otherwise have been necessary.

Thus typical interaction with our profiler involves three programs: `start`, `stop`, and `Parse`. The user executes `start` to begin a profiling session. When he is done running the programs of interest, he executes `stop`. To facilitate interpretation of the resultant output stream, we provide the `Parse` program, which takes as an argument the name of a file written by our sampler. `Parse` produces two similar streams, except that one aggregates on both application and EIP, while the other aggregates on only application. The former lets the user see where particular applications spent their time, while the latter more clearly shows how the various applications divided CPU time.

In addition to the EIP, our user might also like to obtain the actual instruction and the method containing it . Given the program binary, she can easily do so by running the shell command `objdump -d <bin>`. This produces an assembly version of the program organized as a sequence of instructions, each keyed by its corresponding EIP. The `objdump` command also associates groups of instructions with their enclosing method.

We achieved this on-off functionality by making all places where the kernel called our profiling code dependent on a global variable. We then added two system calls, which toggle the value of this conditional. These calls also perform some data structure maintenance, which we will discuss further below. Our `start` and `stop` programs are simply C invocations of these syscalls; users are free to supply additional programmatic interfaces.

2.2 Infrastructure

The method `on_hard_interrupt` comprises the core of our software; the kernel calls it each time the hardware timer expires (ten times per second). This event triggers many activities, which the kernel handles in one of two ways: as a hard or a soft interrupt. Hard interrupt code executes immediately, and no other interrupts may occur until it completes. In contrast, a soft interrupt only uses the hard interrupt to reschedule the actual activity for a later time. To prevent the application from terminating before we sample it, and to permit access to the correct EIP, the entirety of our profiling operation must occur as a hard interrupt.

The body of on_hard_interrupt primarily serves to note the application and corresponding EIP at regular intervals. Doing so requires maintenance of a few data structures, and efficiency further behooves it to provide a batch mechanism for periodically dumping this data to file. Having presented a high-level overview of our strategy, we next explore the details of the data structures we chose to accomplish these goals.

2.3 Appids

The first challenge involves naming. To correctly interpret his results, our user must differentiate different applications, and also distinct instances of the same application, which may run simultaneously. The kernel solves this by using the process ID (PID) to provide unique names. When an application ends, however, the kernel may reuse its PID. Our application cannot allow this, as it must "remember" old applications that have completed, and store their profiling data separately from that of newer applications. We thus introduce the concept of an `appid` to provide unique names through the lifetime of a profiling session. As Figure 1 shows, an appid consists of the `name` of the application's binary as well as an `instance`, which

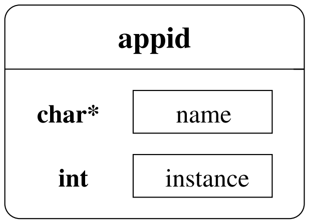

Figure 1: **The appid data structure** *represents a unique name within the context of a sampling session. The name field contains the name of the binary, while the instance field encapsulates the number of times that binary has executed.*

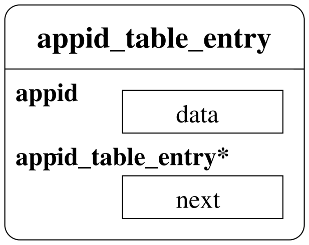

Figure 2: **An appid_table_entry** *represents the largest instance seen for a particular application. Since it is accessed through a linked list, it contains a pointer to the next entry.*

indicates how many times that particular application has run during the current session.

2.4 Max Appid Table

To implement this naming scheme, we must provide a means of generating new appids. Toward this end, we introduce the `max_appid_table`, a hash table probed with an application name. Conflicts create linked lists in the buckets, so we use for each element a struct consisting of a piece of appid `data` and a pointer to the `next` element in the bucket. In this context, the instance field of each appid corresponds to the instance number of the most recently created instance of that application. Figures 2 and 3 show these structures schematically.

The max_appid_table supports three main operations. The kernel calls the most frequently used, `increment`, whenever it needs to generate a new unique name. This function uses `ELFHash` [4] to probe the table for the name of a given application. If found, the code increments

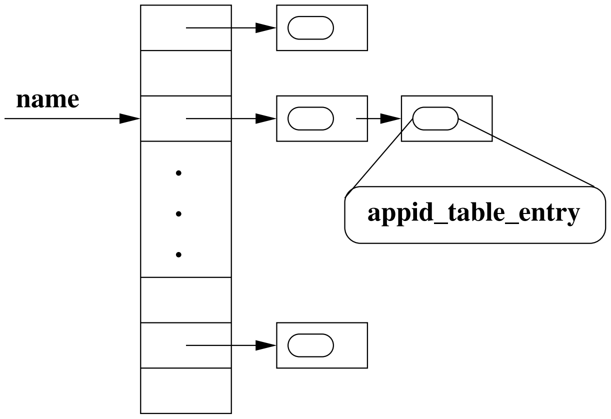

Figure 3: **The max_appid_table** *hashes application names to the largest existing instance number. Synonyms are handled by creating linked lists in conflicting buckets.*

the corresponding instance; otherwise, it adds a new appid to the table, initialized with an instance of zero. Either way, the new instance value is returned for use in constructing a new appid.

The user must also initialize the table before sampling begins; the start system call accomplishes this. Similarly, the stop system call deallocates max_appid_table entries.

2.5 Pid Appid Table

When the timer triggers on_hard_interrupt, the kernel provides two pieces of information: the current PID and the corresponding EIP. Before our code can record this data, it must first convert the PID to an appid. A second hash table, the `pid_appid_table`, orchestrates this. This hash table maps a PID to an appid, again using ELFHash. Like the max_appid_table, the pid_appid_table handles conflicts by generating linked lists. As shown in Figure 4, each element thus contains fields for the `pid`, the corresponding `appid`, and a pointer to the `next` element. Figure 5 shows these entries in the context of the entire table.

The on_hard_interrupt code must always probe the pid_appid_table to effect a PID to appid translation. If the PID does not hit, on_hard_interrupt is sampling a new process for the first time. It thus uses the max_appid_table to generate a new appid to represent this PID incarnation, and inserts this into the pid_appid_table for future refer-

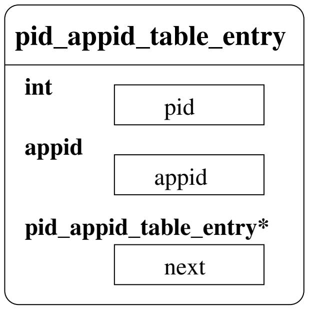

Figure 4: **A pid_appid_table_entry** *associates a PID with an appid. It requires the PID for verifying hits, as multiple PID's may hash to the same bucket. Similarly, the next pointer indicates further conflicting entries, if any.*

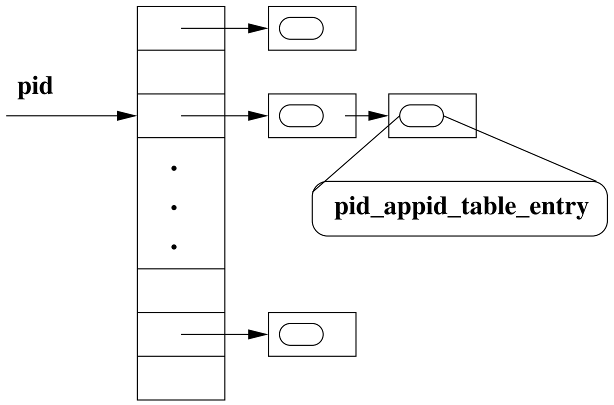

Figure 5: **The pid_appid_table** *maps unique kernel names, PID's, to unique sampling session names, appids, for the lifetime of its component PID's. If the kernel reuses a PID, the pid_appid_table may contain different mappings for the same PID at different times, because the same PID can represent multiple different processes through time.*

ence. To avoid confusing incarnations, we must remove mappings from the pid_appid_table as soon as a process dies and its PID becomes available for reuse. The kernel therefore calls a `pid_appid_table_delete` routine in ???

Like the max_appid_table, the pid_appid_table also requires `init` and `free` functions, which the start and stop system calls invoke, respectively.

2.6 Summary of Kernel Changes

We implemented the majority of the code for the above structures outside of the kernel, and performed integration by merely adding the header files to `linux/include/` and the C files to `linux/kernel`. However, as mentioned previously, some of our code did require changes and additions to existing files. We briefly summarize these here.

First, to implement our start and stop system calls, we added entries to the `sys_call_table` in `linux/include/asm/unistd.h` and call number stubs to `linux/include/asm/unistd.h`. We incorporated the call implementations into one of our original C files, `interface.c`.

Second, while our code creates pid_appid_table entries on demand in on_hard_interrupt, correctness requires us to remove cached values as soon as the kernel can reuse a PID, or on process deletion. Thus we added a call to our `on_process_delete` function, which calls `pid_appid_table_delete` if sampling is enabled, to `do_exit` in `linux/kernel/exit.c`, which handles other process delete functionality.

Finally, we needed the kernel to call on_hard_interrupt on every timer interrupt. The kernel already includes some profiling mechanisms (see Section 5), namely `x86_do_profile` in `linux/include/asm/hw_irq.h`. This function let us distinguish interrupts occurring in kernel space from those is user space. However, determining this condition required use of the argument `regs`. While the variant of 2.4.21 on our personal machines provided this as the argument to x86_do_profile, the version on our Crash and Burn disk did not. Thus we also ended up changing the caller of this function, `smp_local_timer_interrupt` in `arch/i386/kernel/time.c`. This function fortunately had access to regs, so we merely changed the argument to x86_do_profile. In addition to deciding whether or not to sample, we also use regs to obtain the user-level EIP at the time of the interrupt. The final argument that on_hard_interrupt needs, the PID, we access via the `current` struct.

2.7 On Hard Interrupt

We close our discussion of our implementation with a brief outline of the body of on_hard_interrupt, which x86_do_profile passes the current user-level PID and EIP. If the user has not enabled sampling, this method does nothing. Otherwise, it first probes the pid_appid_table with the `pid` argument to determine the corresponding appid. If the probe does not hit, on_hard_interrupt generates a new appid using the max_appid_table, and caches this value in the pid_appid_table for next time. It then utilizes a relic of our attempts at buffering (see Section 7). In essence, on_hard_interrupt only exports data to `/var/log/messages` via `printk` every *n* samples, where *n* is a constant chosen on compilation. For our experiments we used 11, because as a prime, it gave even hash behavior when used as a buffer size, and because it was small enough to cause frequent dumps.

Thus, on each sample, on_hard_interrupt inserts the current sample into a buffer of this size, `active_buf`. When it fills the last slot, it executes a routine to empty the buffer, `background_process`. For this implementation, background_process actually executes within the hard interrupt and merely prints the contents of the buffer to printk, and then re-initializes it. Section 7 explains why the implementation we have just presented does not use real buffering, and also describes the several designs and strategies we attempted.

3 Evaluation

We next evaluate the implementation just described, in terms of correctness and performance. For both aspects, we used three benchmarks.

3.1 Benchmarks

The first, `loop`, contains only a simple for loop that adds the index of the current iteration to a running total for one billion iterations. The second, `call`, has nested for loops. The outer acts as above, except that it only iterates one hundred thousand times, and each iteration adds the result of a procedure call to the running total. The called procedure takes as an argument the current iteration, and performs the same summation of the index to a running total with its argument as the number of inner iterations. Our final benchmark, `btree`, executes for three million iterations. Twenty percent of these perform random inserts into the tree, while the remaining eighty percent perform look-ups. The B-tree itself implements traversals via loops and insertions recursively. Nodes do not have parent pointers.

3.2 Correctness

To determine correctness, we performed three trials for each of the three benchmarks, with sampling enabled. We then copied the applicable section of /var/log/messages into a separate file, and used Parse to aggregate the EIP counts. We also used objdump to obtain an assembly version of each benchmark, for a reality-check.

3.2.1 Loop Benchmark

Figure 6 shows a histogram of the mean counts for each EIP observed across all three loop benchmark trials. We expect sampling of loop to yield only a few distinct EIP's because the code spends most of its time in a single for loop, which should correspond to just a few assembly instructions. Indeed, Figure 6 meets these expectations, as all 2128 samples (counting all three executions) touch only six distinct instructions.

However, this data gains far more heft when combined with knowledge of the actual assembly. The pertinent fragment of the objdump generated from the loop executable follows. We omit the hex translations of the instructions for brevity, and include the section label for context. Note that we also exclude several unreferenced lines from the beginning of the `main` section.

```
...
08048480 <main>:
 ...
 80484a6: mov    %esi,%esi
 80484a8: mov    0xfffffffc(%ebp),%eax
 80484ab: cmp    0xfffffff8(%ebp),%eax
 80484ae: jl     80484b4 <main+0x34>
 80484b0: jmp    80484c4 <main+0x44>
 80484b2: mov    %esi,%esi
 80484b4: mov    0xfffffffc(%ebp),%eax
 80484b7: lea    0xffffffec(%ebp),%edx
```

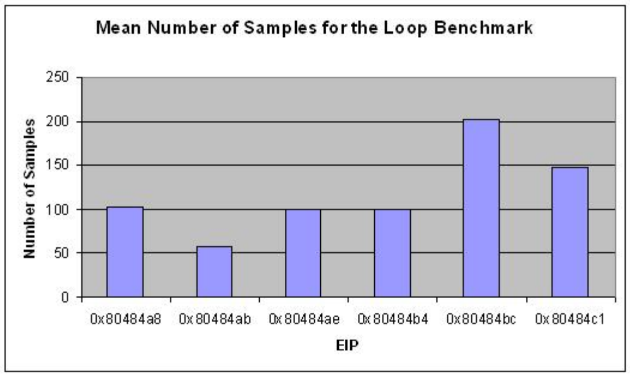

Figure 6: **EIP distribution from sampling the loop benchmark.** *Even though our software took well over seven hundred samples while running this benchmark, all were of one of six EIP's. This strongly supports our thesis that our software samples correctly: since this program spends the majority of its time within a single for loop, we expect only a small range of EIP'S to appear, and these to come frequently.*

```
 80484ba: add    %eax,(%edx)
 80484bc: lea    0xfffffffc(%ebp),%eax
 80484bf: incl   (%eax)
 80484c1: jmp    80484a8 <main+0x28>
 80484c3: 90                       nop
...
```

Most importantly, this code shows that all measured EIP's do occur in the `main` section of the disassembly; correctness demands this as no other section in such a simple program could contain programmer-defined code. Second we see that all six measured instructions occur between the destination of a `jmp` and the actual jmp instruction. This strongly suggests we do indeed observe the part of the code that actually comprises the loop. For comparison, the corresponding C fragment follows.

```
for (index = 0; index < MAX_ITERATION;
     index++) {
  sum += index;
}
```

3.2.2 Call Benchmark

We performed an analogous experiment for the slightly more complex benchmark, call. Figure 7 shows the his-

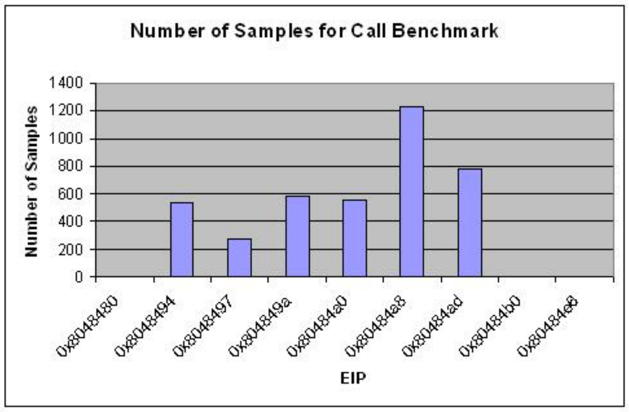

Figure 7: **EIP distribution from sampling the call benchmark.** *This histogram shows the mean number of references for each bucket among the three trials. The call benchmark showed more variance among its trials than loop did; there were three EIP's that were measured only once, and only in one benchmark. Averaged, then, these buckets were empty. This explains the presence of the empty buckets in the graph.*

togram. Here we see more variation among the trials. The buckets that appear empty in the figure each held only a single sample, and that from only one benchmark. (These anomalies were distributed across the benchmarks, however). Like loop, on the other hand, samples remain concentrated among a few EIP's.

To better understand this data, we next examine the corresponding disassembly. We first discuss instructions sampled from the `loop` routine.

```
08048480 <loop>:
 8048480: push   %ebp
 8048481: mov    %esp,%ebp
 8048483: sub    $0x8,%esp
 8048486: movl   $0x0,0xfffffff8(%ebp)
 804848d: movl   $0x0,0xfffffffc(%ebp)
 8048494: mov    0xfffffffc(%ebp),%eax
 8048497: cmp    0x8(%ebp),%eax
 804849a: jl     80484a0 <loop+0x20>
 804849c: jmp    80484b0 <loop+0x30>
 804849e: mov    %esi,%esi
 80484a0: mov    0xfffffffc(%ebp),%eax
 80484a3: lea    0xfffffff8(%ebp),%edx
 80484a6: add    %eax,(%edx)
 80484a8: lea    0xfffffffc(%ebp),%eax
 80484ab: incl   (%eax)
 80484ad: jmp    8048494 <loop+0x14>
 80484af: nop
```

The first "empty" EIP refers to the first instruction of the loop function, the procedure called inside the outer loop. The next six buckets all correspond to instructions within this function. Close interpretation of the assembly again reveals that these instructions comprise the major work of the code: in this case, the loop within the procedure call. The second anamolous bucket also references part of the loop function, but in this case a piece outside of the loop. The final single sample belongs to main, within the outer loop.

```
080484b8 <main>:
 ...
 80484e3: cmp    0xfffffff8(%ebp),%eax
 80484e6: jl     80484ec <main+0x34>
 80484e8: jmp    8048508 <main+0x50>
 ...
 80484f2: call   8048480 <loop>
 ...
 8048504: jmp    80484e0 <main+0x28>
 ...
```

The fact that nearly all of the samples were of instructions comprising the inner loop certainly attests to the correctness of our implementation. At first, it may seem surprising that we observe so few samples of the outer loop. However, consider that the inner loop executes five billion iterations, while the outer loop executes only one hundred thousand. Given this perspective, the observed ratio of two out of 11,843 seems more than reasonable. To help the reader better understand these ratios, we present the two loop bodies featured in the benchmark.

```
int loop (int j) {
  int i;
  int sum = 0;
  for (i = 0; i < j; i++) {
    sum += i;
  }
  return sum;
}
...
for (index = 0; index < MAX_ITERATION;
     index++) {
sum += loop(index);
}
```

3.3 B-Tree

Finally, we examine the btree benchmark; Figure 8 shows the histogram. Our presentation of the btree data differs significantly from that of loop and call, mostly as a result of the benchmark's size: btree references over two hundred distinct instructions. Thus, instead of bucketing by EIP, we instead bucket by the encompassing function. Across all three trials, we see samples of instructions in seven different functions. About ten percent occur in main, which is responsible for deciding whether to perform a lookup or an insert, and which must also generate random input for the inserts. The lookup routine itself takes about twenty percent of the samples; it iterates through levels of the tree. Insert itself has no samples, but we expect this, as insert merely calls a recursive insert variant. That has no samples, either, however, but `find_child` and `get_parent_key` together took about fifteen percent of the samples, and only insert utilizes these routines. This distribution does not appear to follow the proportional distribution of lookups and inserts performed; we attribute this to the fact that inserts cost far more CPU time than do lookups. Finally, `find_index` takes the most hits, at about fifty percent. As both lookup and insert require several calls to this function, these results seem quite reasonable in terms of the code structure. Like call, btree features one anomaly: a few samples to the plt section, which the compiler automatically inserts into all executables.

During the course of instrumenting this data (placing EIPs into function buckets) we found that all EIPs that we explicitly examined in fact matched exact entries in the corresponding binary. Beyond the more qualitative results we have discussed so far, this adds significant credence to our claims of correctness. This results directly from the variable instruction size of the x86 architecture. Since instructions may vary from one to seventeen bytes and are not aligned, the probability of our sampler exactly matching instruction addresses were it incorrect is negligibly small.

Thus far our sampler appears flawless. However, ex-

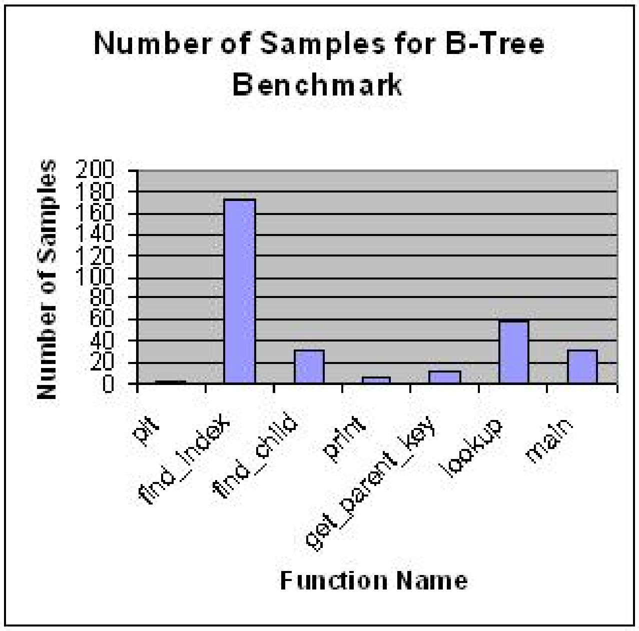

Figure 8: **EIP distribution from sampling the btree benchmark.** *This histogram again shows the mean number of references for each bucket among the three trials. However, since the btree code has many more lines than that of call, for this graph we group code by function, instead of by EIP. The names mostly explain themselves; plt is section inserted by the compiler at the beginning of all assembly files.*

amining the btree samples in detail did reveal some unexplained profiles. Sixty-five percent of the btree samples, on average, occurred to a set of drastically different, unknown addresses. (Note that the percentages above were with respect to known addresses, and that these unmapped addresses were excluded from the histogram.) While all addresses that we sampled in loop, call, and the rest of btree begin with 0x804 and are seven hex digits total, these addresses begin with 0x400 and are eight hex digits. Our only hypothesis is that our profiler somehow sampled system code and attributed it to btree. This seems reasonable because many of the other applications we sampled (X and gnome, for instance) had addresses of this length beginning with 0x4.

	Control	Sampling	% Difference	Backup	% Difference
loop	10.477351	10.476556	−0.008	10.800774	3.087
call	38.906508	38.902799	−0.010	40.228744	3.383
btree	63.732426	63.181360	−0.865	67.768769	6.333

Figure 9: **Profiler Performance.** *This table gives the execution time in seconds of the three benchmarks loop, call, and btree, in three configurations. Control refers to execution without sampling enabled, sampling refers to sampling enabled but backup commented out of the build, and backup refers to sampling enabled with periodic data dumps included. The left percent difference column compares sampling to control, while the right one compares backup to control. A negative percent difference indicates a performance loss relative to the control experiment. The data given is the median of the three runs taken for each of the nine configurations.*

3.4 Performance

Having convinced ourselves of the correctness of our implementation, we next turn to performance evaluation. We performed this experiment by measuring the execution time of all three benchmarks with sampling, with sampling but without dumping the data, and without sampling. As before, we took three runs for each configuration. Figure 9 shows the results. First we compare the performance of the control (sampling disabled) to that of the configuration described in Section 2: periodically on_hard_interrupt writes the buffered samples to /var/log/messages via printk. We see that loop and call have fairly low overhead at about three percent, but btree has significantly higher overhead, at six percent. Interestingly, an earlier run of the same set of experiments showed btree with a significantly lower overhead, at less than one percent. The latter results make far more sense: since btree is less CPU-intensive, it should show less overhead as less execution occurs in user-space and less sampling should therefore occur. Indeed the EIP results above would support this hypothesis, as while btree runs nearly twice has long as call, it has less than half as many samples, even including the unmapped ones. We can only conclude that the system is quite sensitive to variation, and that a larger set of tests should be run to gain confidence.

Placing faith in these results nonetheless, we explore the source of these overheads in more detail by eliminating the most expensive operation in on_hard_interrupt:

dumping the samples to file. The left-hand percent difference column of Figure 9 shows the results. Strangely, our measurements show a consistent performance improvement over the base configuration across all three benchmarks. However, the actual values are so similar, that we hypothesize the difference between the two configurations is simply too small to be measured. The variation within a single set of trials (e.g., loop with sampling disabled) was more than the difference between trials (e.g. loop without sampling versus loop without the data dump). Again, a larger body of trials would probably shed light on this situation.

Finally, we draw attention to the fact that we cannot evaluate performance by measuring the number of sampling events. This is because the kernel does not call our code if an interrupt occurs in kernel space. Thus a drop in user-level samples does not imply an increase in sampling overhead, because an increased percentage of kernel execution would have the same effect.

To summarize, performance measurements show significant variation. Generally, we believe that our profiling method causes relatively little overhead (around three percent) and that nearly all of this may be attributed to the dumping mechanism. Thus to reduce overhead with the current configuration, the user could trade memory space by arbitrarily increasing the size of the buffer. This would not affect correctness as the sample stop system call automatically performs a final dump before clearing the buffer. We have also explored techniques to rotate buffers and to perform backups outside the hard interrupt via another thread or process. Section 7 summarizes our strategies and findings.

4 Future Work

The preliminary results have been promising but there are a lot of improvements that could be implemented for a much more efficient and useful analysis tool. The immediate optimizations to improve SLEEPY are described below.

First off, the evaluation and results of sampling with backup are worst case scenarios. An improvement on single threading is naturally multithreading so that multiple buffers could be used. Backup could be differed to a later time and the hard interrupt would be shorter. This scheme would also allow for interesting results regarding what amount of buffering would be optimal for such a system. Further results would be to compare our implementation to similar profilers such as OProfile or Prospect /citetsariounov2002.

Other useful optimizations would be beneficial for usability purposes. A MySQL front end would allow programmers familiar with database operations to easily query the samples. In order to allow for instruction level granularity, instruction translation should be implemented during the timer interrupt. Similarly, queries on the percentage of time spent within all the instructions of a method could result in interesting results. A software back end could be written to map the EIP to the system map of functions on a binary file using nm or some other mapping utility. Another idea is to incorporate our user-level profiling with the kernel profiling that is already present in Linux. Distribution-wise, it would be cleaner to extract our code from the kernel itself and turn it into a module.

5 Related Work

System profiling has been the focus of several studies. Specifically, system calls, processor performance counters, and sampling have been the methods for determining system and application bottlenecks. Using this as a starting point, we implement our own system in the x86 architecture.

When attempting to determine the source of delay in an application one of the first ideas is I/O. Therefore, if the system calls of all running applications can be caught and determined. In particular, Parrot and Bypass are transparent user-level middleware projects, which simplify the creation of interposition agents between the operating system and the CPU so that system calls can be recorded [6][7]. In addition, by measuring the time each request was issued and completed, one can gain an accurate picture of where an application is spending its time in relation to the rest of the system. However, such a system limits the profiling to system I/O usage.

Other system profilers use processor chip counters. The Brink and Abyss suite allows various performance events to be measured by accessing the performance counters on the Pentium 4 on a per application basis [5].

VTune is Intel's answer for profiling on their line of processors [1]. It is a system-wide profiler and it is not able to easily separate a single process' information since other applications are updating the performance counters at the same time in a sampling period. While performance counters are provided with the processors themselves, their implementation has proven to be buggy and the errors in their execution are undocumented.

Other profilers bypass performance counters entirely and sample the information systematically. DCPI interprets the instruction pointer (EIP) on performance counter overflows [2]. The sampled EIP must be shifted six cycles into the past for the current EIP. DCPI is also specific to the Alpha architecture, which is a very clean architecture with a fixed length instruction set. Meanwhile our profiling is specific to the x86, a variable length instruction architecture. Instead of creating using a performance counter interrupt, we utilized the timer interrupt, which occurred every 10 milliseconds. Therefore the PC did not have to be shifted to accurately "interpret" the current instruction.

The most similar implementation of profiling is OProfile, the Linux system-wide profiler based on DCPI [3]. OProfile first attempts to use performance counters. If unavailable it defaults to sampling the EIP on timer interrupts. Again, our implementation is based entirely on sampling the EIP, but our samples are taken within the kernel, not as a module, so our sampling should incur less overhead.

6 Conclusions

We have presented SLEEPY, a mechanism for continuously profiling user-level applications when running Red Hat Linux on an x86 architecture. Our method incurs relatively low sampling overhead (about three percent with the configuration described in this paper) and can be almost completely eliminated by increasing the size of an in-kernel buffer. We have also created user-level software that permits enabling and disabling of sampling without changing the kernel, as well as programs to help interpret the data generated by our kernel code. One drawback of our implementation is that it requires modifying and recompiling the kernel. However, it can easily be extended to also perform kernel sampling.

Perhaps the most valuable lesson we learned during this project was the significance of disabling interrupts. Since the main body of our code occurs during a hard interrupt, when no other interrupts may occur, we struggled quite a bit with finding a way to add a background process to our implementation. Unfortunately, none of efforts ever did pan out. We also learned the value of finding and taking advantage of existing tools and resources: much of what we achieved would not have been possible without the significant body of unpublished research so readily available through the Internet.

For further lessons, please see Section 8.

7 Unused (But Attempted) Research Ideas

The main problem encountered is due to working during the top half of the timer interrupt. The purpose of using the top half is the ability to sample the EIP from the previously running process before the its status is saved and swapped out during a context switch. The problem lies in the fact that certain activities are prevented from running during this period.

Our original algorithm for storing the samples involved a clean buffering scheme. This algorithm depended on having the ability to create a kernel thread to service a buffer once it is full. This kernel thread would execute outside of the timer interrupt, shortening the length of the timer interrupt. However, it was discovered upon implementation that kernel threads are not allowed in an interrupt.

Therefore, other means were explored to avoid kernel threads. Tasklets looked to be a good solution for scheduling an activity to be deferred. In order to use tasklets kernel code had to be separated in order to prevent recursive dependencies on code required by the profiling in the hard interrupt. The main idea was to associate a tasklet to each buffer, including the tasklet as a data member in the buf_pool_element data structure. This caused problems since our code relied on interrupt.h while another header file in the kernel (hw_irq.h) relied on sample_buf.h as well as interrupt.h.

After implementing a tasklet outside of the interrupt to ensure it worked, the scheduling and enabling was moved inside of the hard interrupt. But during sampling the task never occurred. Again, the ability to defer backing up of buffered samples was infeasible through tasklets.

However, we will still present our algorithm for buffering, which was also implemented, but not present in the results. At the beginning of the sampling period an array of buffers is preallocated. Once a buffer is full, it is marked as full and not used again until the samples it contains is backed up. Then the buffer with the smallest index in the array is set as the active buffer. If all the buffers fill up samples are dropped (not overwritten for simplicity). Given the state where all the buffers are full, a sample is taken and the array of buffers is searched to see if a buffer has been freed. Therefore, dropped samples are minimized. More interestingly, this allows for easy viewing of the largest buffer that has been utilized to determine the buffer requirements by the system. Statistics can also be taken on the average number of buffers needed as emptying buffers in a differed fashion is dependent on scheduling.

In addition to backup issues, we also wanted to gather more information during on_hard_interrupt. In particular, we wanted to provide a means of dereferencing the EIP to obtain the hex for the instruction. We could then translate this into assembly during the background process, largely eliminating the need to use objdump manually. We did in fact devise an assembly routine to accomplish this, but like our buffering exploits, what worked outside of on_hard_interrupt caused the system hang when placed inside. Before abandoning this feature, we hypothesized that virtual memory mappings perhaps do not always endure, thus causing some kind of access error when dereferencing the EIP. We further hoped to include information about the enclosing function. Had we added these two pieces of data, the following point would have been of even greater interest.

We currently provide a Java program to parse the dumps made by our sampler. However, a MySQL interface that lets the user perform aggregations herself on tuples of interest would make the data we gather far more useful. Toward accomplishing this, we did install MySQL on our Crash and Burn disk, but we never got around to working on accessing a database programmatically with C. The end result would have been similar to what we presented in our evaluation, but much of the labor could have been automated away.

Figure 10: **The Crash and Burn Lab is very cold.**

8 Final Comments

The memory management comment was noteworthy , but memorable at a later time when we needed the information and remembered the comment. From Linus himself: “Fork is rather simple, once you get the hang of it, but the memory management can be a bitch.” See ’mm/memory.c’: ’copy_page_range()’

It seems as though the comments in the kernel are very descriptive as to the feelings of those who put their life and soul into the kernel. A graph would be nice if we are interested in this: Fuck 35 Bitch 15 Shit 27

The crash and burn lab is extremely cold. Every crash and burn disk should come with a space heater. See Figure 10.

The girl who works nights at Quick Bite scoops up the largest ”two” scoop waffle cones we have ever seen.

References

[1] Intel Corporation. Vtune. http://www.intel.com/software/products/vtune/.

[2] J. Anderson et al. Continuous profiling: Where have all the cycles gone? *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, pages 1–14, October 1997.

[3] J. Levon. Oprofile. http://oprofile.sourceforge.net/about/.

[4] Aaron Partow. http://www.partow.net.

[5] B. Sprunt. Managing the complexity of performance monitoring hardware: The brink and abyss approach, 2004.

[6] D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, pages 4:39–47, 2001.

[7] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. *Proceedings of the Workshop on Adaptive Grid Middleware, New Orleans,* September 2003.