Simple and General Statistical Profiling with PCT

Charles Blake
Laboratory for Computer Science
Massachusettes Institute of Technology
cb@mit.edu

Steve Bauer
Laboratory for Computer Science
Massachusettes Institute of Technology
bauer@mit.edu

Abstract

The Profile Collection Toolkit (PCT) provides a novel generalized CPU profiling facility. PCT enables arbitrarily late profiling activation and arbitrarily early report generation. PCT usually requires no re-compilation, re-linking, or even re-starting of programs. Profiling reports gracefully degrade with available debugging data.

PCT uses its debugger controller, `dbctl`, to drive a debugger's control over a process. `dbctl` has a configuration language that allows users to specify context-specific debugger commands. These commands can sample general program state, such as call stacks and function parameters.

For systems or situations with poor debugger support, PCT provides several other portable and flexible collection methods. PCT can track most program code, including code in shared libraries and late-loaded shared objects. On Linux, PCT can seamlessly merge kernel CPU time profiles with user-level CPU profiles to create whole system reports.

1 Introduction

Profiling is the art and science of understanding program performance. There are two main families of profiling techniques, automatic code instrumentation and statistical sampling. Code instrumentation approaches use a high-level language compiler or linker to incorporate new instructions into object file outputs. These instructions count how many times various parts of a program get executed. Some instrumentation systems [12] count function activations while others [1, 21] count more fine-grained control flow transitions. Sampling approaches momentarily suspend programs to sample execution state, such as the value of the program counter. How frequently certain locations occur during an execution estimates the relative fraction of time incurred by those parts of the program.

PCT is a sampling-based profiling system that shows a new way to construct effective performance investigation tools. PCT demonstrates that the same tools programmers are familiar with for answering questions about correctness can be used for effective performance analysis. The philosophy of PCT is that profiling is a particular type of debugging and that the same preparations should be adequate. The focus of PCT is CPU-time profiling rather than real-time profiling, though, in principle, sampling may be applied to either.

PCT is also flexible and easy to use. Enabling profile collection rarely requires re-compiling, re-linking, or even re-starting a program. In its simplest usage, adding a one word prefix to the command-line can activate collection over entire process subtrees and emit a basic analysis report at the end. PCT can track CPU time spent in the main program text, shared libraries, late-loaded dynamic objects and in kernel code on Linux. PCT works with a variety of programming languages.

A novel aspect of PCT is that it allows sampling semantically rich data such as function call stacks, function parameters, local or global variables, CPU registers, or other execution context. This rich data collection capability is achieved via a debugger-controller program, `dbctl`. Using debuggers to probe program state allows PCT to sample a wide variety of values. Statistical patterns in these values may explain program performance. For example, statistically typical values of a function's parameters may explain *why* a program spends a lot of time in that function.

Additionally, `dbctl` can drive parallel non-

Feature	Description
Causally Informative	Maximize ability to explain performance characteristics
Extensible	Sample many kinds of user-defined program state
Late-binding	Defer as long as possible the decision of whether to profile
Early-reporting	Make profile reports available as soon as possible
Non-invasive	Require no extra program build steps or copies of objects
Low-overhead	Minimize extra program run time
Portable	Support informative profiles on any OS and CPU
Robust	Do not rely on program correctness, in particular clean exits
Tolerant	Report quality should gracefully degrade with worse system support, poorer profile data, and less rich debugging data in object files.
Exhaustive	Track as much relevant CPU activity as possible
Multilingual	Support different programming languages, multi-language environments

Table 1: Desirable profiling system features.

interactive debugging sessions. As the original process creates children, `dbctl` can spawn off new debugger instances, reliably attaching them to those children. Using a debugger-controller allows a *portable* implementation of process subtree execution tracing tools, such as function call tracers.

The functionality of PCT gracefully degrades with the available support in the system and the executables of interest. Debugging data or symbol tables are needed for highly meaningful reports. Nevertheless even stripped binaries allow some analysis. For example, one can track the usage of dynamic library functions or emit annotated disassembly. In concert with instrumentation-based basic block-level profiling such as `gcov`, PCT can even estimate CPU cycles per instruction. Sampled data can be windowed in time to isolate different CPU intensive periods of a program's execution. The various report formats are available through a set of composable primitive programs and shell pipelines.

Profile reports may be generated at any time, even prior to program termination and several times over the life of one process. Several granularities are available for data aggregation and report formats. Depending on the debugging data available in executables, users can select how to display program locations. This may be at the level of individual instructions, line numbers, functions, source files, or even whole object files or libraries.

The organization of this paper is as follows. Section 2 discusses our design objectives. To make PCT's capabilities more concrete, Section 3 shows a few examples. Section 4 then elaborates upon PCT's implementation of data collection. Section 5 details report generation strategies. Section 6 evaluates the overhead and accuracy of the toolkit. Section 7 discusses some other approaches to profiling. Section 8 describes how to obtain the software. Finally, Section 9 concludes.

2 Design Objectives

The design goals of PCT were driven by user needs and the inadequacies or inaccessibility of prior systems. Table 1 highlights these objectives. The following section argues for the importance of each in turn, and the approach of PCT in general.

Programmers use profiling systems to understand what causes performance characteristics. E.g., if certain functions dominate an execution, then a profile should tell us why those calls are made, and why they might be slow. If functions are called with arguments implying quite different "job sizes", then a profile should be able to capture this for analysis. Exactly how ***causally informative*** profiling can or should be is an open issue. More information is better up until some point where overhead and analysis tractability concerns become a problem. Programmers currently have far more *a priori* knowledge about what to look for than any automatic system can hope to have. A practical answer is an ***extensible collection*** system that lets users decide what program variables are most relevant to subsequent performance analysis.

Performance problems often arise only on inputs by end users unanticipated by the programmer or in very late stages of testing. These issues are thus discovered at the worst possible time for rebuilding a program and all its dependencies. Long running programs such as system services often have phased behavior. That is, sections of the program with quite distinct performance characteristics execute over various windows in time. Profiles over entire program executions can introduce unwanted averaging over this phased behavior, making results more difficult to interpret. A direct and flexible way to address this problem is to allow ***late-binding***. Ideally, activating and deactivating profile collection should be possible at *any* stage in the life cycle of a program. As an immediate correspondent, ***early-reporting*** is also desirable so that long running programs with highly active phases do not need to terminate before a profile can be examined. Together, these let programmers apply whatever knowledge they have about phased behavior.

Classic instrumentation techniques raise a number of administrative, theoretical, and practical issues. Instrumentation usually requires extra steps to build two versions of executables and libraries, ordinary and instrumented. It is often problematic to require recompilation of all objects in all libraries or to require commercial vendors to provide multiple versions of their libraries. Providing multiple library versions can be a burden even on the various contemporary open source platforms. For instance, profiling instrumented libraries in `/usr/lib` on open source distributions are scarce or entirely absent. Also, it is possible to instrument code long after linking it. For example, binary rewriting techniques along the lines of Pixie [23] or Quantify [14] allow this. Completely dynamic instrumentation is also possible.[18]

Nonetheless, instrumentation, at whatever time, raises several issues. Instrumented code really is not the same as the original code. Subtle microarchitectural effects can make it hard to understand the overhead of new instructions. Beyond theoretical accuracy issues, there is also a more practical concern in that getting the instrumentation correct is a challenging problem in itself. Profiling instrumentation can interact badly with "new" compiler features, optimization strategies, or uncommon language usage patterns. In the worst case, which is all too frequent, the produced executable may not even run correctly. Finally, with the possible exception of fully dynamic instrumentation, this strategy is inherently less extensible. Only *a priori* data types can be extracted, and this is usually limited to simple counts of executions to avoid re-implementing a good deal of compiler technology.

While instrumentation has the virtue of precision, the above considerations suggest that we should go as far as possible with systems that are ***non-invasive*** to the stream of instructions the CPU encounters. In essence, this implies a sampling-based approach. Sampling also has the virtue of incurring tunably ***low overhead***.

Performance problems often arise only when programs are used in very different environments from where they were developed. Platform-specific profiling packages can be more efficient and occasionally more capable. However, they do not help if performance problems cannot be reproduced on supported platforms or environments. Programmers also have a rational resistance to learning and relying upon multiple, disparate system-specific tools and interfaces. Therefore, a more ***portable*** system is more valuable.

Portability concerns also suggest a sampling approach. Any preemptive multi-tasking OS already suspends and resumes programs as a matter of course. The only missing pieces for profile collection are a means to suspend frequently and a mechanism to inspect the state of a program. Reading a program's state is inherently simpler than re-writing its code. Thus, sampling is typically no more intrusive than ordinary preemption and requires simpler, less specialized system support than automatic instrumentation.

Some past profiling systems have used in-core buffers that are written to disk in `atexit()` handlers at the end of a clean program shutdown. A system should not mandate clean termination in order to diagnose performance problems. One *phase* of a program may warrant performance investigation even if other phases are buggy. Inputs needed to trigger performance pathologies may also instigate incorrect behavior. Conversely, performance pathologies can easily trigger failure modes not ordinarily encountered. Thus, profile collection should ideally be ***robust*** against program failure. Many existing implementations could be adapted to be more cautious in this regard.

Bottleneck code can potentially hide anywhere in a program. Restricting profiling coverage to only code compiled or linked into the address space in certain ways leads to many "holes" in the accounting of where execution time was spent. The more ***exhaus-***

tive code coverage is, the more likely a profile will unravel performance mysteries.

Mixed programming language systems have become pervasive in modern software development environments. While C and C++ are a canonical example, it can be the case that a wider range of languages, e.g. FORTRAN, Ada are supported within one program. If unified source-level debugging exists for these ***multilingual*** environments then source-level profiling should also be supported. A profiling system wedded to a particular programming language or code generation system is too inflexible.

PCT is the first profiling system known to the authors to possess *all* of these properties simultaneously. Extensible and informative data collection is achieved through the ability of source-level debuggers to compute arbitrary expressions and do detailed investigation of program state. PCT is non-invasive and low-overhead since sampling does little more than what the OS ordinarily does during task switching. The sampling rate can be changed to trade-off overhead with accuracy. Late-binding is achieved by delaying activation of sampling code or having it instigated by an entirely different process, namely the debugger. Portability derives from relying only on old, well-propagated system facilities dating back to the mid-1980s. Robustness ensues from the earliest possible commitment of data to the OS buffer cache, which is closely related to producing reports as soon as any data has been collected. The system is as multi-lingual as the executable linking environment allows. PCT is as exhaustive as the debugger, OS, and build environment allows. The very small report generation modules enable tolerating various levels of debugging data, customizing reports, and porting PCT to a new platform.

PCT is also a small system. The code for PCT is only 3,500 commented lines of C code and 300 lines of shell scripts. This compact delivery of functionality is possible only because PCT greatly leverages common system facilities.

3 Examples

On many systems getting a quick profile is as simple as: `profile myprogram args...`

More concretely:

```
$ profile ./fingerprint /bin | head
13.9%   /u/cblake/hashfp/binPoly64.C:101
 7.6%   /u/cblake/hashfp/binPoly64.C:86
 7.3%   /lib/libc-2.1.2.so:getc
 5.3%   /u/cblake/hashfp/fingerprint.C:112
 4.3%   /u/cblake/hashfp/binPoly64.C:80
 4.2%   /u/cblake/hashfp/binPoly64.C:70
 4.0%   /u/cblake/hashfp/binPoly64.C:96
 3.7%   /u/cblake/hashfp/binPoly64.C:102
 3.0%   /u/cblake/hashfp/fingerprint.C:116
 2.8%   /u/cblake/hashfp/fingerprint.C:104
```

By default line numbers are used for source coordinates. If only symbols are available they are used. Finally, raw *objectfile:address* pairs are printed when there is no debugging data at all.

Profiling mixed kernel and user code on Linux is similar. Below is a quick profile of the disk usage utility which recurses down a directory tree summing up file allocations:[1]

```
$ profile -k du -s /disk/pa0 | head
30.4%   /usr/src/linux/vmlinux:iget4
12.4%   /usr/src/linux/vmlinux:ext2_find_entry
 5.4%   /usr/src/linux/vmlinux:try_to_free_inodes
 3.5%   /usr/src/linux/vmlinux:ext2_read_inode
 3.3%   /usr/src/linux/vmlinux:unplug_device
 2.4%   /usr/src/linux/vmlinux:lookup_dentry
 2.0%   /usr/src/linux/vmlinux:system_call
 1.8%   /usr/src/linux/vmlinux:getblk
 1.0%   /lib/libc-2.1.2.so:open
 0.9%   /lib/libc-2.1.2.so:__lxstat64
```

Note the call hierarchy in the following program:

```
int worker(unsigned n) { while (n--) /**/ ; }
int dispatch_1(unsigned a) { worker(a); }
int dispatch_2(unsigned b) { worker(b); }

int main(int ac, char **av) {
    dispatch_1(10000000);
    dispatch_2(20000000);
    return 0;
}
```

Before doing any profiling it is obvious that essentially all run time is in the function `worker()`. There are two paths to this function, as shown clearly via the debugger-based hierarchical profile:

```
$ profile -gdb -l3 hier-test
67.6%   worker  <-  dispatch_2  <-  main
32.4%   worker  <-  dispatch_1  <-  main
```

[1]Directory and i-node data was pre-read to make the results reflect CPU time spent in the 2.2 kernel.

Finally, consider sampling more semantically rich data. In general this requires amending a 10 line `dbctl` script similar to the following:

```
 1 EXEC() ".*" {
 2 #include "gdbprof_prologue.dbctl"
 3  PAT_GROUP(default) {
 4   PAT(1) "signal=\"SIGVTALRM\"" | OUT("pc") {
 5              "backtrace 4" | OUT("stack");
 6              # OTHER DEBUGGER EXPRESSIONS
 7              "continue";
 8          }
 9 #include "gdbprof_epilogue.dbctl"
10  }
11 }
```

A full description of the pattern-driven state machine language is beyond the scope of this paper. The main `EXEC` pattern on line 1 restricts which executables the entire rule applies to. Lines 4, 5, and 6 simply capture the pc in the output file "pc", and four levels of stack backtrace in the output file "stack". Adding more debugger commands and data files is just a matter of adding a line to the `dbctl` script. Depending on the expressions sampled, various post-processing steps may be needed.

We provide a simpler interface for the common case of sampling scalar numbers. Below shows how to sample values of `n`, inside the function `worker()` where it is meaningful.

```
$ profile -gdb \
     -expr 'hier-test@worker@n' 'int-avg' \
     hier-test
8.38e+06
```

The `-expr` option takes two arguments – a context-specific expression to generate data in the debugger, and a program to format the collected data. The context specific expression is an '@' separated tuple of strings: a program pattern, a function pattern, a debugger expression.

4 Data Collection

The general implementation philosophy of PCT is to support a full set of options for every aspect of profiling. This minimizes the chance that some system limitation will prevent any profiling outright and enables "best effort" profiling. Small, composable primitives also ease tailoring PCT behavior. Basic users generally use several generic driver scripts, while more advanced users create their own tailored script wrappers.

4.1 Activating Sampling Code

PCT has three basic collection strategies: debuggers, timer signal handlers, and `profil()`.[5] The first never requires re-starting or re-linking a program, but can have substantial real-time overhead. The latter two are fall-back, library-based strategies which can be used when low overhead is preferrable or when debugger support is inadequate. The library-based samplers are, however, less portable. They require linker support for C++-style global initializers and also require either a dynamic library pre-loading facility or manual re-linking. Dynamic pre-loading is commonly available with modern dynamic linkers [3], though not all programs are dynamically linked. In the worst case, if C++-style linking is unavailable, programmers can manually invoke the initializer inside their main() routine. We now examine these samplers in more detail.

The oldest portable profiling primitive is `profil()`. This system call directs kernel-resident code to accumulate a histogram of program counter locations in a user-provided buffer. While the call interface does potentially allow multiple executable regions, the authors know of no operating systems that can activate more than one region at a time. To ensure robustness, PCT allocates the userspace buffer as an `mmap()`-ed file. This also allows the profile to be accessible at any time to other processes, such as report generators. The `profil()` call to activate kernel-driven collection can be done with either a dynamically pre-loaded or statically linked-in library.

A source-level debugger affords a more general sampling activation strategy. The debugger uses the `ptrace()` facility and catches all signals delivered to the process, including virtual timer alarms. `ptrace()` can be used to attach and detach from processes at any time and any number of times over the lifetime of a process. PCT uses a debugger-controller program to implement this procedure portably.

The PCT debugger controller drives the debugger which in turn controls the process via `ptrace()`. The debugger calls the POSIX `setitimer()` system call in the context of the target process. This installs virtual time interval timers for the target process. Once these timers are installed, the kernel will deliver VTALRM signals periodically to the target process. At each signal delivery, control will be transferred to the debugger. At this point the debugger driver issues whatever debugger commands are necessary to collect informative data and then continue program execution. For example a backtrace or where command typically produces a sample of function call stack data. The real time of the sample can also be recorded. Section 4.3 discusses the details of debugger control.

When library code can be used, a pre-main() initializer sets up interval timers, signal handling, and data files. gcc-specific, C++, or system-dependent library section techniques can be used to install the library initializer. Library code may be statically or dynamically linked, or preloaded for dynamically-linked executables via the $LD_PRELOAD environment variable.

PCT collection behavior is controlled through the $PCT environment variable. It controls options such as output directories, histogram granularity, data format, and so on. It also provides a convenient switch for whether profiling happens at all. $PCT and $LD_PRELOAD can both be inherited across fork() and exec(). This conveniently enables profile collection activation on whole process subtrees.

4.2 Collecting Data

4.2.1 Types of Code

The debugger collector supports tracing whatever code the debugger can recognize. All debuggers handle the main program text. Most modern debuggers, e.g. gdb, can debug code in shared libraries and late-loaded object files on most operating systems.

PCT library-based collectors have more specific restrictions. They can collect data on several kinds of code:

- One contiguous region – usually the main program text. This is the oldest style of profiling and works in almost any OS and scenario.
- Shared libraries. On Linux the instantaneous bindings of virtual memory regions to files are exposed. Reading /proc/*PID*/maps reveals executable regions and corresponding object files.

  On most BSD OSes ldd reports load addresses of shared libraries. These addresses may be cached in files similar to /proc/*PID*/maps and read in by the global initializer.
- Late-loaded (e.g. dlopen()ed) code: On Linux whenever a PC cannot be mapped to a known memory region, the signal handler re-scans /proc/*PID*/maps to attempt to discover new regions. If it succeeds, the region table is updated and the counts processed. When an object file for the PC cannot be found, further re-scanning is inhibited to suppress repetitive failed searches.
- Kernel code: Linux provides a /proc/profile buffer for the main text of the kernel. Currently, Linux does not support profiling loadable kernel modules.

As mentioned in Section 4, profil()-based collection is generally only available for a single contiguous region of address space. These other types of code are all supported by the more general library-based collector. Profiling kernel modules could be added to Linux or other OS's using the same techniques that PCT uses for managing shared libraries and late-loaded code.

4.2.2 Types of Profiling Data

Collection methods based on profil() or /proc/profile afford little choice as to the type or format of data collected. Other sampling methods, such as $LD_PRELOAD and debuggers allow collecting a variety of data. This flexibility creates choices as to *what* data is collected for later analysis, how and where it is stored.

PCT provides several data storage formats. The specific profiling situation will usually determine which is best. The choices are:

- Debugger output files: a different log file is used to save the output of each user-specified sampling expression.
- Histogram file: stores frequency counts for various code regions. This guarantees bounded space, but cannot window data.

- Sample-ordered log file: allows simple time-windowing of collection events, post facto histograms, but can grow in size indefinitely.

- Circular log file: This is similar to the sample-ordering except that the user bounds the size, which effectively saves only the last N samples.

4.3 Controlling Debuggers

The PCT toolkit includes a controller program `dbctl` for driving debugger tools such as `gdb`. The controller program is a state machine described by user specified files. A transition in the state diagram occurs when the controller recognizes a regular expression in the output of the debugger. For each transition, there is a set of debugger commands to issue as well as a series of controller actions. The debugger commands can be any command appropriate for the debugger tool being controlled. For example, controller actions might include logging debugger output, spawning new debuggers to attach to child processes, and capturing specified debugger outputs in internal controller variables.

In the case of profiling, the `dbctl` tool sets up the interval timers in the processes to be profiled. When the timers expire a signal is raised which transfers control to the debugger and generates output indicating the context of suspension. The controller recognizes various process contexts and issues context-specific instructions. These can include writing out the call stack, local variables and function arguments, or any arbitrary debugger expression.

Profiling is not the only application of `dbctl`. The tool can also be used to implement a portable `strace` [7] or `ltrace` [11] facility. While many debuggers have function call tracing capabilities, tracing an entire *process tree* is more challenging.

The UNIX `ptrace()` mechanism has traditionally had rather weak support for following both a parent and child process across a `fork()`. Typically, there is a constraint of one-to-one binding between a traced process and a tracing process. After a `fork()` only one of the potentially traced processes can remain under external control. The other is released to be scheduled by the OS. Breakpoints left in a untraced process cause a SIGTRAP that causes the process to die since it has no debugger to catch the signal on its behalf. Therefore a debugger arranges things so that breakpoints are disabled across a `fork()` for one of the two processes.

For `dbctl` this means that if we arrange to follow the parent, then a `fork()`ed child could "run away" from the controller, possibly `fork()`-ing grandchildren before `dbctl` can attach a new debugger to it. Similarly, if we arrange to follow the child, the parent could run away forking other children before a new debugger can be attached. Ideally, kernels would provide a standard interface to "fork and suspend" `ptrace()`d processes. Lacking a natural interface to avoid this race condition, PCT developed an interesting work around. Our `fork()`-following protocol guarantees that no child process is ever lost and that breakpoints can be re-enabled immediately after the call to `fork()`.

The protocol works as follows. First, we set a breakpoint at all `fork()` calls to catch the spawning of children. When the `fork()` breakpoint is hit, the debugger disables all breakpoints so that the untraced process will not get spurious `SIGTRAP` signals. It then installs `pause()` as a signal handler for `SIGTRAP`. Finally, it sets a breakpoint for the instruction following the call to `fork()`.

The parent remains `ptrace()`d all along, and immediately traps to the debugger because of the `fork()`-return breakpoint. All normal breakpoints are re-enabled. The child process id can be found on the stack as the return value from the `fork()`. `dbctl` uses this pid to attach a new instance of the debugger to the child process.

Concurrently, the `fork()`-return breakpoint causes a `SIGTRAP` to be delivered to the child. Since it no longer has a tracing process, the process' own signal handler is invoked. In this case that function is `pause`, a system call which simply waits until some signal is delivered. When a newly spawned debugger successfully attaches to the process it interrupts the ongoing `pause`. The procedure of attaching a new debugger is now complete. `dbctl` then re-establishes any necessary breakpoints and so on in both processes and lets them run again.

4.4 Limitations

Library-based histogram collectors face a problem with `fork()`d processes/threads which truly run in parallel (i.e. on multiple CPU systems). The parallel processes can potentially overwrite each other's counter updates. The result is a missed counter increment with the last writer winning the counter bump. This is rare and is probably not an issue in practice. In any event, one can assess the number of lost counts by the total scheduling time given and the total counts collected. If there is a major discrepancy one can switch to log-file profile collection which does not share this problem.

Hierarchical samples are currently only supported with the debugger collector. The code to walk back a stack frame is conceivably simple enough for some CPUs to embed directly into our signal handler library. This could drastically reduce overhead at the cost of sacrificing some CPU portability and probably some language neutrality.

Our debugger controller requires that executables either be dynamically linked to the C library, or if statically linked contain a few critical symbols, such as `pause`, that may not be strictly required to be present. Of course the debugger can also do very little with executables stripped of all symbol data.

Sampling rates are limited by maximum `VTALRM` delivery rates. These typically range from 1 to 10 ms. Some systems allow increasing this rate. Depending on the richness of collected data, it may not be desirable to increase this rate, as that would entail more real-time overhead.

5 Data Analysis

PCT data collection strategies produce files with quite different information. Designing one monolithic way of reducing this data to programmer interpretable relationships is hard. Instead PCT provides a toolkit of data aggregation and transformation programs. These can be easily composed via UNIX shell pipelines. Their usage is simple enough that users can tailor simple scripts toward individual circumstances and preferences.

5.1 Source Coordinate Resolution

Debuggers emit high-level source coordinates as a matter of course. They are constrained by how much debugging data was been compiled into the executables and libraries being used. If these object files have been compiled with the full complement of debugging data then source and line number-level, function-level, and address-level coordinates are all available. Usually, all are present in the textual debugger output PCT records. This mode of data collection then yields a lot of choices. PCT lets users select the coordinates to be used in reports.

On the other hand, library-based collectors do not resolve program counter addresses to source coordinates while the program is running. Instead they record only program counter addresses and defer higher-level coordinate translation to report generation-time. These addresses are saved in compact binary data formats that keep logs small and minimize IO overhead. There is usually one binary data file for each independently mapped region of program address space. Embedded within these files are the path names and in-memory offsets of the memory-mapped object files. This provides the key information for deferred translation to understand how addresses in memory correspond to addresses in the object files.

Translation of program addresses to more meaningful source coordinates can be awkward. Object file formats vary substantially. The GNU binary file descriptor library gives some relief, allowing the writing of programs which directly access debugging data and symbol tables. This library may be unavailable, out of date, or not support the necessary object file formats. As the examples in Section 3 show, PCT makes a best-effort attempts to translate coordinates.

At the least, if executable files retain their symbol table, the system `nm` program or the debugger can interpret it. If there is no debugger, the *GNU binutils* package provides a convenient `addr2line` program which can map PCs to file:line source coordinates. If there is a debugger installed, then a debugger script can operate just as `addr2line` in the restricted capacity of address translation. If there is no debugging data at all, as for stripped binaries, then a disassembly procedure is always an option. Indeed, for instruction-level optimization, it may even be desired to produce count-annotated disassembly files as reports. Of course, assembly-level expertise is required to interpret such reports.

PCT provides a printing program `pct-pr` which bridges the gap between PCT binary data files and programs which affect address translation. `pct-pr` assumes a simple and convenient protocol for shell pipeline syntax. Translators are run as co-processes to `pct-pr`. I.e., programs read a series of PCs on their standard input and emit corresponding source coordinates to their standard output.

This co-process setup allows fine-tuning of the protocol with `pct-pr` command-line arguments. For example, `printf()`-style format strings allow tailoring the object file PC stream to input requirements, and also allow customizing the output stream. Users can stamp PCT binary data files with particular PC translation requirements, or simply describe which translators to use on the `pct-pr` command line.

PCT also provides a program `addr2nm` to translate PCs using only the system `nm` and symbol tables in executables. This program first dumps `nm` output into a cache directory if it is not already there. Once this file exists, `addr2nm` does an approximate binary search on each inbound PC, discovering the symbol with the greatest lower address.

5.2 Data Aggregation

One often wants to aggregate profile data over various uses of the same programs or libraries. The canonical example is combining many runs of short-lived programs. In the context of profiling a process tree, one may want to examine many distinct processes as one aggregate set of counts. For example, a `libc` developer might be interested in all the usages of some particular function, e.g. `printf()`, throughout a process tree. PCT supports these various styles of aggregation via simple filename conventions and traditional Unix filename patterns. A user can select collections of data files by common filename substrings such as the name of the object files of interest. For example, `pct-pr /tmp/pct/ct/myprogram.*/libc*` would generate source coordinates for all samples of the `libc` code used by `myprogram`.

The output of source coordinate translation for each file in a collection is a simple pair of sample counts and labels for that location in the program. The granularity of these labels, e.g. function or source:linenumber, will determine the notion of similarity for later tabulation. This stream can be sorted with both PCT-specific and standard Unix filters to produce a stream where text lines referring to "similar" code locations are adjacent. A filter can then aggregate counts over text lines with these "similar" suffixes, and hence the similarity determines the level of aggregation. These aggregates are effectively histograms of program counter samples over the address space of the programs. The histogram bins are determined by the granularity of labels. E.g., function-granularity source coordinates will result in a report of the time spent in various functions.

Users often find it easier to think about time fractions rather than raw sample counts. PCT supports this with a filter that totals the whole text stream and then re-emits it with counts converted to percentages. These percentages are normalized to whatever particular selection of counts is under consideration – either over multiple runs or over multiple objects or other combinations.

The interface for users to these capabilities are simplified by simple shell script wrappers. For example, `pct sym% /tmp/pct/ct/myprogram.*/libm*` will create a function-level profile of time spent in the math library.

There are a few PCT report styles that provide more context around the sampled program locations. PCT can create entire copies of source-code files annotated with either counts or time fractions. We also have an Emacs mode much like `grep-mode` or `compile-mode` to drive examination of *filename*:*line number* profile reports. This mode allows a user to select report lines of high time fraction and automatically loads a buffer and warps the cursor to that spot in the code.

6 Evaluation

A few concerns arise in evaluating any profiling system. First, one must ask if profiling overhead is obtrusive relative to real-time events. Large overhead could make results inaccurate. Bearing in mind that programs being profiled may be quite slow, excessive real-time overhead could dissuade programmers from using the system. A second concern is the accuracy of profiling numbers produced by the system. The following subsections discuss these issues.

6.1 Overhead

The overhead of any sampling system is tunably small (or large). There is a fundamental overhead-accuracy trade-off. The more frequently samples are taken, the more overhead incurred by interrupting the program and recording the samples. However, the larger the sample rate the more accurate a picture one acquires in a given amount of time.

Large sample rates may be desirable. Some programs run only briefly, but are CPU intensive while they run. Accurate probabilities may also motivate a fast sample rate.

On modern CPUs, the cost of signal delivery and resumption of execution system call is typically less than 20 μsec. Thus library-based sampling procedures are very low overhead.

We have measured a `gdb`-based sampling as taking 500..1000 μsec on 700..1300 MHz Pentium III and Athlon-based systems running Linux, FreeBSD, and OpenBSD. The precise time varies depending on the complexity of parameter lists being decoded, the depth of the stack, the efficiency of the OS, and the CPU. However almost all of this overhead is `gdb` making many calls to `ptrace()`to reconstruct the argument lists of functions in the backtrace. It is possible to arrange a more minimally informative `gdb` sampling which does no address translation or decoding. This resulted in under 100 μsec, including the round-trip context switch.

These numbers are still relatively encouraging. For 10 ms sampling granularities the overhead is almost unnoticeably small unless quite rich samples are being taken. At 1 ms sampling rates the overhead starts to become near a factor of two, but overhead is not prohibitive until near 100 μsec rates. This also suggests that a debugging library could result in substantial overhead reduction by computing only the necessary output. Alternately, debugger features could control the verbosity of output more finely.

Also note that, at least on uniprocessors, it does not matter how many processes are being traced. Only one process has the CPU at a time. So some fractional overhead applies to the real time consumed by the entire system of processes.

6.2 Accuracy

Sampling overhead does not directly impact the accuracy of time estimates. Any constant amount of sampling overhead has the effect of simply increasing the sampling period. Hence the *variance* of time spent handling signals and recording samples might impact the accuracy of program counts. In practice, with code that has known time fractions, sampling time variance seems to have a negligible effect.

Assessing precise time fractions of a program creates a need for large sample sizes. The statistics of location counts are approximately binomial. The program is suspended at some location, i, with probability p_i. The mean of a binomial random variable for N trials each of which has probability p_i is simply Np_i, while the standard deviation is $\sqrt{Np_i(1-p_i)}$. The frequency, $p_i = n_i/N$ thus has a fractional error proportional to $1/\sqrt{N}$. Suppose, for example, location 1 has count n_1 and location 2 has count n_2. It is easy to show that a two standard deviation test for the condition $p_1 > p_2$ is approximately $n_1 - n_2 > 2\sqrt{n_1 + n_2}$. E.g., to decide $p_1 > p_2$ for $n_2 = 1$ requires $n_1 \geq 5$. Fortunately, precise probabilities are usually less important than just identifying the expensive areas of a computation.

Reduced real-time performance is the most significant down-side of overhead. For very high sampling rates and very rich data collection a program can run much slower than in its native mode.

Correlations between the time of sampling and paths in the program are more problematic. Consider the specific example of a function which takes almost exactly as long to execute as the time between timer expirations. Also assume this function is repeatedly invoked and dominates the execution time. It should be clear that the program will always be suspended near the same location. Computation is distributed over all the code implementing this function. [17] discusses this problem in the context of CPU usage statistics. DCPI [8] addresses this problem by randomizing the size of the time interval between samples.

While `profil()` is inflexible in this regard, any other PCT collection method optionally uses one-shot timers and re-installs timers with randomly spaced delays in the alarm handler. Using large multiples of a typical 10 ms time quantum are likely to seriously reduce the achieved sample size. However provided that successive periods are unpredictable, even a random alternation between 10 ms and 20 ms guards against a little accidental synchronization. The fixed underlying timer clock makes truly preventing synchronization effects difficult. One cannot build a random interval from even random multiples of a coarse intervals. Synchronization at the scale of the underlying time quantum could still cause problems. Truly random sample-to-sample intervals clearly require specialized OS support.

7 Related Work

Profiling is as old an art as writing programs. As with debugging, there has been a large amount of tool-building and research devoted to automating tasks that were originally done with hand-coded instrumentation. These prior profiling systems all take a more narrow view of profiling than the PCT philosophy of profiling as a *type of debugging* in which programmers apply the same familiar tools and preparations.

Automatic instrumentation introduced the possibility of collecting richer data than classic `profil()`-style samples, such as dynamic call graphs, basic block activations, and control flow arc transitions. Increasing complexity of software systems has driven a need for multi-process and even whole system profiling systems. Both static and dynamic profile-driven optimization have become a focus for those interested in performance.

Additionally, some have considered limited notions of *higher-level* profiling [22] such as the implementation of abstract data types or other alternative algorithm selection. This approach instrumented programs to record, for example, where data is typically inserted into an ordered list. It used this data to decide between array or linked-list representations. Inserts at the beginning favor linked representations while those at the end favor arrays. PCT might be leveraged to answer similar questions without modifying the program to use an instrumented data structure library.

There have been a large number of compile-time automatic instrumentation systems, all of which are invasive, early-bound, and non-extensible. An early hierarchical profiler was `gprof`.[12] The programs `tcov` [6] and `gcov` [1] are similar to `gprof`, but instrument basic blocks instead of function calls. Many implementations of these types of profiler are not robust to improper program exits and are not tolerant of inadequate data in some objects.

Many systems have also implemented some form of link-time instrumentation or post-link-time binary re-writing.[23, 14, 9, 15, 24, 20] These address rebuilding issues somewhat and have some weak extensibility. A significant invasion of foreign code may remain, though. The code must be inserted to count executions, or, in more involved cases, log procedure arguments.

Recently, a number of researchers have begun investigating the limits of *dynamic* instrumentation – the re-writing of running executables. IBM has a system called DProbes [18, 10] which enables generic kernel-based late-bound instrumentation. This is similar to our debugger-controller based approach, but aims to build up a toolset for various machines and architectures rather than relying on the existing debugger infrastructure. Much lower overhead would likely be possible via this approach, but a great deal more work would need to be done to allow the sort of arbitrary expressions collectible with debuggers.

Beyond instrumentation systems, there has been significant prior progress in PC-sampling-style profiling as well. There is the classic `prof` [4], and many latter-day counterparts. The most sophisticated system along these lines is probably the Digital Continuous Profiling Infrastructure (DCPI).[8] DCPI has focuses on understanding how the microarchitectural features of Alpha processors play out in full system applications. This system is unfortunately proprietary and non-portable as many of its most impressive features rely upon CPU and OS support. The focus on low-level CPU behavior instead of high-level semantics of programs makes this system more useful for compiler writers and other assembly-level optimizations. For instance it does not support hierarchical call path samples along the lines of.[13]

SGI's `IRIX`-specific SpeedShop [2] system is probably the closest system in spirit to PCT, though it stops short of a full debugger-profiler. It does sampled hierarchical profiling, has graceful report degradation, and is late-binding. However, in addition to being specific to `IRIX` on MIPS, it is restricted to dynamically linked executables, and fails to be exhaustive in terms of kernel-resident and late-loaded code. Being proprietary, it is difficult to evaluate its extensibility.

There have been many kernel-level profiling tools as well. Some system call tracers like `strace -c` support simple system call profiling.[7] Yaghmour's Linux Trace Toolkit [25] is a useful kernel-level event monitoring facility. PCT can leverage easily accessible `/proc/profile` data on Linux. PCT's ptrace()-based techniques do not easily extend to kernel code since the necessary process control features are not typically available on the kernel itself.

The type of non-interactive debugging PCT does is similar to the Expect system for controlling interaction.[16] Our debugger-controller is similar, but with configuration syntax tailored to profiling and the ability to handle large subtrees of processes simultaneously. dbctl also does not rely on the relatively slow logic and string processing of Tcl.[19] As noted earlier, dbctl also allows non-interactive process control other than state sampling.

8 Availability

PCT is freely available under an open source license. More information and current software releases can be obtained at the PCT web page:

http://pdos.lcs.mit.edu/~cblake/pct

In the realm of simple profiling, every UNIX after AT&T version 7 has support for profil() functionality, which provides at least some capability. Kernel profile integration is currently only available on Linux. The ldd command on OpenBSD and FreeBSD is informative enough to allow tracking shared library usage and process tree profiling.

Generalized profiling should be available on any system with a good source-level debugger for the programming languages of interest. Currently, gdb works well on the above mentioned systems as well as Solaris, HP-UX 9,10,11, AIX, Irix, SunOS, various other BSD's and probably many more platforms. Ports of our interaction scripts to dbx and other debuggers are under way.

9 Conclusion

Each year software systems grow in complexity from multiple code regions per address space to multiprocess programs. Correctness becomes harder to achieve, and conventional wisdom is to postpone performance analysis as long as possible. PCT requires no more preparation than for debugging. This allows programmers to interleave the optimization and debugging of their program however they see fit.

PCT unifies access to a number of existing profiling features that have been available for some time and extends profiling in new directions. The PCT debugger-based profiling architecture substantially extends the sort of data that automatic profiling can collect. One can sample programmer-definable, context specific data. Such samples can often more readily expose higher-level algorithmic issues, such as a mismatch between program structures and user inputs. The overhead of PCT scales reasonably with the complexity of the program data being sampled and with sampling rates.

Finally, PCT is very portable by design, requiring no special CPU or OS features or support. Informative data can be gathered on most code in flexible ways. Reports can be generated flexibly based on various data aggregations while the program is still running. These features usually require no recompiling, relinking, or even re-starting of users' programs.

References

[1] gcov: a test coverage program. http://gcc.gnu.org/onlinedocs/gcc-3.0.

[2] Irix 6.5 speedshop user's guide. http://techpubs.sgi.com/.

[3] ld(1). Unix man pages.

[4] prof(1). AT&T Bell Laboratories, Murray Hill, N.J., UNIX Programmer's Manual, January 1979.

[5] profil(2). Unix man pages.

[6] tcov(1). AT&T Bell Laboratories, Murray Hill, N.J., UNIX Programmer's Manual, January 1979.

[7] W. Akkerman. strace home page. http://www.liacs.nl/~wichert/strace/.

[8] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), Nov. 1997.

[9] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[10] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[11] B. Driehuis. ltrace home page. `http://utopia.knoware.nl/users/driehuis/`.

[12] S. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982.

[13] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the Summer 1993 USENIX Conference: June 21–25, 1993, Cincinnati, Ohio, USA*, pages 1–13, Berkeley, CA, USA, Summer 1993. USENIX.

[14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991.

[15] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, 18–21 June 1995.

[16] D. Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the USENIX Summer 1990 Technical Conference*, pages 183–192, Berkeley, CA, USA, June 1990. Usenix Association.

[17] S. McCanne and C. Torek. A randomized sampling clock for CPU utilization estimation and code profiling. In *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 387–394, Berkeley, CA, USA, Winter 1993. USENIX Association.

[18] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track (FREENIX-01)*, pages 297–308, Berkeley, CA, June 2001. The USENIX Association.

[19] J. K. Ousterhout. Tcl: An embedable command language. In *Proceedings of the USENIX Association Winter Conference*, 1990.

[20] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 1–7, Berkeley, CA, USA, Aug. 1997. USENIX.

[21] A. D. Samples. Profile-driven compilation. Technical Report UCB//CSD-91-627, UC Berkeley, Department of Computer Science, 1991.

[22] A. D. Samples. Compiler implementation of ADTs using profile data. 641, 1992.

[23] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, Computer System Lab, Nov. 1991.

[24] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, June 1994. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI).

[25] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 13–26, Berkeley, CA, June 18–23 2000. USENIX Association.