CPPROFJ: Aspect-capable Call Path Profiling of Multi-threaded Java Applications

Robert J. Hall
AT&T Labs Research
180 Park Ave, Bldg 103
Florham Park, NJ 07932
bob-3CpprofJ02-@channels.research.att.com

Abstract

A primary goal of program performance understanding tools is to focus the user's attention directly on optimization opportunities where significant cost savings may be found. Optimization opportunities fall into (at least) three broad categories: the call context *of a general component may obviate the need for some of its generality; cross-cutting program* aspects *may be implemented suboptimally for the particular context of use; and* thread dependencies *may cause unintended delays. This paper enhances prior work in call path profiling[5] in several ways. First, it provides two different call path oriented views on program performance, a* server *view and a* thread *view. The former helps one optimize for throughput, while the latter is useful for optimizing thread latency. The views incorporate a* typed time *notation for representing different program activities, such as monitor wait and thread preemption times. Second, the new framework allows* aspect-oriented *program profiling, even when the original program was not designed in an aspect oriented fashion. Finally, the approach is implemented in a tool,* CPPROFJ, *an aspect-capable call path profiler for Java. It exploits recent developments in the Java APIs to achieve accurate and portable sampling-based profiling. Three case studies illustrate its use.*

1. Introduction

Understanding the performance of a program is an essential prerequisite to effective program optimization and system re-engineering. One wants some assurance in advance that a proposed change to the code will result in a significant improvement, because ineffective code changes result in lost development and test time and risk introducing new bugs. For complex systems, such understanding can only come from profiling tools applied during representative runs of the system. A central goal of such tools should, therefore, be to attribute run time costs to *optimizable* design decisions in the source code. The tool can then focus the developer's attention on the most significant optimization opportunities first. And note that pertinent performance information is valuable as input to other tools as well, such as compilers and other program optimization tools.

Where, then, do we find optimization opportunities? There are at least three broad categories of them. First, optimizations are often enabled by *context of use*. A common case is when a general component is used in a context that does not require its full generality, so a less general version can be used which incurs less cost. Schematically,

```
Application function
   --calls--> general component
       --calls--> utility routine
```

A well known example of this is copy elimination, where a tool can remove calls to a copy operation within a general component, because it can prove that when the general component is called from the context of the particular application function, the copying is unnecessary for correctness.

Another fertile hunting ground for optimizations is the set of design choices made in implementing *abstract data types and other cross-cutting program aspects*. Such opportunities arise when some program aspect is implemented inadequately to handle the distribution of input cases seen by the deployed system. For example, an abstract *Set* type implemented as a linked list, while most appropriate for small *Set*s, will not scale well to large sets, where an implementation using a hash table may work better.

Finally, a third source of performance bottlenecks in modern applications is *thread contention and other inter-thread dependencies.* For example, an input/output processing thread may be stalled waiting for a thread marked with higher priority, even though its data has arrived from the input device. It can often be better to mark the i/o processor as higher priority so that it can start the next read or write before letting the more compute-bound threads continue.

Existing program performance tools, while providing a large amount of low-level data, do not provide *direct, integrated* support for detecting these types of optimization opportunities. Thus, this work automates the tedious and error prone process of inferring useful optimization opportunities from low level data. In this paper, I describe a new tool for multi-threaded Java programs called CPPROFJ. It builds conceptually on the original *Call Path Profiler* work[6, 5], which showed how to directly discover costs incurred by routines in the various call contexts in which they are used. The novel contributions of this paper include the following.

- I extend call path profiling to handle multi-threaded server-style Java applications through the definition of two views: the *Thread View*, which profiles costs incurred by instances of a single thread class, and the *Server View*, which profiles CPU costs of all threads. The former is most useful for improving *thread latency* (start to finish time for a thread), while the latter is most useful for improving *system throughput* (thread completions per unit time). This multi-view call path approach supports search for the first category of optimization mentioned above.

- Cross-cutting program *aspects*[2, 9] incur costs within contexts as well as do program methods. The current work allows one to define aspects after the fact and reports costs on them in the same way as for program methods. This is true even if the original program is not implemented in an explicit aspect-oriented language. This addresses the second category of optimization opportunity discussed above.

- The extension to multithreaded applications necessitates introduction of *typed time costs*. The Thread View profiles report (a) time spent when actually running, (b) time spent sharing the CPU with equal priority threads, (c) time spent waiting while higher priority threads are running, (d) time spent waiting for a monitor, and (e) idle time waiting for other types of events. While the notion of typed time costs is not new, its integration with arbitrary call context is; this allows finding costs due to inter-thread dependencies in arbitrarily deep contexts, addressing the third category of optimization opportunity mentioned above.

- I extend previous work on call path refinement profiles by defining a new refinement profile type, the *upward split refinement*, which gives one the ability to work "upwards" within an arbitrary call context from lower level routines to find which intermediate callers are incurring the most cost (see Section 2).

- Finally, CPPROFJ exploits the recent development of a debugging API for Java[8] in order to collect the raw data, and so is portable across virtual machines.

This paper is structured as follows. The next section defines call paths, their costs, and describes refinement profiles. These allow the user to control information display and help guide a prioritized search for significant bottlenecks. The following section extends call path profiling to handle program aspects, while the one after that defines the Thread View, the Server View, and typed time costs. After that is a brief exposition of the CPPROFJ tool and its implementation, including how to collect data by stack sampling and how to compute the profiles themselves. Final sections summarize case studies, related work, and limitations.

2. Call Paths and Refinements

CPPROFJ captures the notion of optimization context through the concept of a call path. The atoms making up call paths are simply the methods of the Java program. (In other languages these are known as procedures, functions, or subroutines.) A *call specification* (or *call spec*) represents the set of times during program execution when the call stack of the executing system contains a call from its *caller* method to its *callee* method. A call spec may be either *immediate* or *extended*. An immediate call spec from A to B corresponds to an actual call directly from method A to method B on the runtime stack. An extended call spec from A to B represents all time spent when method A has called zero or more intervening methods, the final of which has called B (which may have called further submethods).

A *call path* represents the set of times during program execution when the call stack of the system satisfies an ordered sequence of call specs. Intuitively, this represents the time spent when a particular nested sequence of calls is executing. For example, `(A B C ..D E)` denotes a call path having (in order down the call stack) a direct call from A to B, a direct call from B to C, an extended call from C to D, and a direct call from D to E. When all the calls of call path P are present on the run time stack of a program at time t in order, we say the execution *satisfies* P at t. Define the *cost* of call path P during an execution of the program to be simply the fraction of the total run time of the execution during which the execution stack satisfied P.

Recursion complicates our understanding a bit, because it introduces a potential ambiguity. For example, does the following stack trace satisfy call path `(A B C)`?

```
A B D E B C
```

By the above definition, it does (because each sequential call is present on the stack), but the question is whether this interpretation is more useful than the more strict interpretation which would require that the callee of the `(A B)` call be identical to (i.e. the same stack frame as) the caller of the `(B C)` call. Call paths do (and should) have the broader meaning, because our goal is to report the costs of optimization opportunities. If we optimize by altering the version of C called from B within calls from A, then it will affect all such calls, even those having intervening callers between the call from A to B and that from B to C. Thus, the costs we report will better reflect potential cost savings due to the optimization opportunity. This difference of interpretation is an important advantage of a call path profiler over stack tree profilers[12] (see Section 7).

A *refinement* profile of a call path P is a (typically, ordered) list of call paths and their costs, where all the call paths are related to P in a particular manner. The most familiar type is the *immediate downward refinement*, which consists of all call paths (having nonzero costs) of the form Px, all immediate callees of the call path P. For example, let P be the call path `(A B ..C D)`; then the immediate downward refinement profile of P could be (for a given program and execution)

```
0.23 (A B ..C D x)
0.04 (A B ..C D y)
0.02 (A B ..C D z)
```

where the cost of each is reported on the left. This type of refinement gives the user information as to how a call path breaks up its time among its immediate subtasks.

An *extended downward refinement* of P consists of all call paths (again, with nonzero costs) of the form $P..x$. The extended downward refinement of `(A B ..C D)` could be

```
0.23 (A B ..C D ..x)
0.21 (A B ..C D ..f)
0.04 (A B ..C D ..y)
0.02 (A B ..C D ..z)
0.01 (A B ..C D ..g)
```

where `f` and `g` are indirect callees of P. This type of refinement gives the user information as to which utility routines and other many-use resources are used the most by aggregating their times over the many uses.

While the downward refinements give information about subtasks, the dual operation is also useful, that of asking which of a call path's callers (within a defined context) are causing it to cost signficantly. For this, CPPROFJ defines the *upward split refinement* profile. Given a call path profile P of the form $P_1..P_2$, the upward split refinement of P at P_2 is the collection of all call paths having the form $P_1..xP_2$ and their (nonzero) costs. As a concrete example, the upward split refinement of `(A B ..C D)` at `(C D)` could be

```
0.15 (A B ..x C D)
0.10 (A B ..y C D)
0.04 (A B ..z C D)
```

This enables a bottom-up approach to resource usage; for example, one can find which contexts are spending the most time in database operations, i/o, or other resource usage. This generalizes the upward refinement profiles of [5], in that the immediate upward refinement is equivalent to an upward split refinement with empty upper call path (P_1).

To illustrate the power of context and upward split refinements, consider this excerpt from a Server View (Section 4) profile of the Email system case study (Section 6):

```
1.000 *
0.281   ..sun.io.CharToByteConverter
...
```

(Note that the `*` aspect, see Section 3, simply denotes time spent in any thread, i.e. all the cpu time of the execution.) This shows that, annoyingly, 28.1% of the cpu time of the server is spent converting Java's unicode characters into ASCII bytes (for subsequent writing to files or to the network). There are valid reasons for doing this in some circumstances, for example when writing a database file, whose contents are created via String manipulations, to disk. But there may be less valid reasons.

By repeatedly applying upward split refinements to this call path, we reach the profile shown in Figure 1. (Note: throughout I will use the abbreviation "`j`" in place of "`java`" in profiles. Also, due to the length of Java method names, I will represent call paths using tree structured indentation as is commonly seen in file system browsers in MacOS and Windows. The cost on a line is for the call path whose *last* method is on that line.) Figure 1 shows that character conversion costs most (13.5%) when writing message body files. This may represent an optimization opportunity: message bodies start out as bytes, so don't convert them to characters unless necessary. On the other hand, the 10.8% potential savings within the database property list save routine is less likely to be avoidable, since the database contents are produced through heavy use of string manipulations. The conversion done when writing out *header* files is insignificant, so there is no point in bothering with it.

3. Aspects

Aspect oriented programming is a way of conceptualizing the design of a program so that individual cross-cutting concerns, such as database access, abstract data type implementations, or input/output operations, are made explicit and implemented in a localized, declarative way. The intuition is that we wish to capture design decisions governing potentially widely scattered code fragments in a central declaration so that they can be more easily maintained and evolved. Of course, a prime motivation for doing this is that when we later learn the decision was made suboptimally,

```
1.000 *
0.212 ..tools.dbplist.save
0.116   j.io.PrintWriter.close
0.108     j.io.BufferedWriter.close
0.108       j.io.BufferedWriter.flush
0.108         j.io.OutputStreamWriter.write
0.108           sun.io.CharToByteConverter
0.173 ..tools.msg.write_bfile
0.135   j.io.PrintWriter.close
0.135     j.io.BufferedWriter.close
0.135       j.io.BufferedWriter.flush
0.135         j.io.OutputStreamWriter.write
0.135           sun.io.CharToByteConverter
0.038 ..tools.msg.write_hfile
0.008   j.io.PrintWriter.close
0.008     j.io.BufferedWriter.close
0.008       j.io.BufferedWriter.flush
0.008         j.io.OutputStreamWriter.write
0.008           sun.io.CharToByteConverter
```

Figure 1. A Server View profile after repeated upward splitting starting from the char-to-byte conversion routine.

we can change it. When the suboptimality is due to inefficiency, we obviously need a way to measure the costs and discover these optimization opportunities.

We can incorporate aspects into call path profiling by defining *pseudo-methods*, which represent a logical disjunction of some set of methods in the program. Examples from the aspect-oriented literature include `C.*` where `C` is a class, and `*.debug*`, representing all methods on all classes for printing debugging information. We say that a true (non-pseudo-) method *belongs to* a pseudo-method A if it matches A's defining disjunction (pattern). We extend the definition of call path to include pseudo-methods by allowing them to appear in call path notations and augmenting our definition of call path costs. We first augment our definition of call specification to include calls to and from pseudo-methods, and then augment what it means for a call stack to satisfy a call specification as follows. (Here, A and B represent pseudo-methods.)

- A stack trace satisfies a call spec of form $(x \quad A)$, where x is a true method, if and only if it has an immediate call from x to a true method t which belongs to A.
- A stack trace satisfies a call spec of form $(x \quad ..A)$, where x is a true method, if and only if it has an extended call from x to true method t belonging to A.
- A stack trace satisfies a call spec of form $(A \quad x)$ if and only if x is a true method that belongs to A and it contains a call to x.
- A stack trace satisfies a call spec of form $(A \quad ..x)$, where x is a true method, if and only if it contains an extended call to x from true method t belonging to A.
- A stack trace satisfies a call spec of form $(A \quad ..B)$ if and only if it contains an extended call from some true method t_A belonging to A to a true method t_B belonging to B.

This augmented notion of call spec satisfaction extends what it means for a stack trace to satisfy a call path containing pseudo-methods as well. Thus, the cost of a call path containing pseudo-methods is then simply the fraction of total time spent when the stack trace satisfies the call path, as before. Note that we do not allow immediate calls from one pseudo-method to another. This can lead to a useless proliferation of profile entries.

The refinement profiles must be extended to incorporate pseudo-methods as well. Given the above definition of meaning and costs of call paths having pseudo-methods as entries, the definitions of the refinements remain the same, except that none of the results will contain immediate calls from one pseudo-method to another, since this case is disallowed by the definition.

The primary usefulness of pseudo-methods in profiles is that they aggregate (possibly overlapping) times spent in different members of the aspect. As an example, in the email system case study (see Section 6), I defined an aspect corresponding to the ADT representing queues of email messages: `msg_queue.*`. This profile fragment:

```
1.000 SMTP_sender
0.051   ..msg_queue.*
```

showed that the `msg_queue.*` aspect (ADT implementation) cost 5.1% of the run time of the SMTP sender threads. This turns out to have been because of a quadratic implementation of the queue that was starting to become a problem due to testing under higher loads (hence longer queues). An immediate downward refinement showed more detail:

```
1.000 SMTP_sender
0.051   ..msg_queue.*
0.034      msg_queue.add_msg
0.015      msg_queue.length
0.002      msg_queue.delete
```

I was able to replace the implementation so that adding messages and finding the length of the queue were constant time operations. This savings is significant when even longer queues occur under heavier loads. Note that the use of the extended call spec involving the `msg_queue` aspect allowed aggregating the time in the various methods so that they appear as significant. Otherwise, their times would likely have been much harder to spot. The addition of a bit of design knowledge (the notion that `msg_queue` is a single aspect) helped find a relatively subtle performance problem during performance testing, rather than having to find it when the load became higher in production use.

4. Views and Typed Time

The complex multi-threaded applications of today may be optimized with respect to many different performance parameters. Two common ones are *latency* and *throughput*. The former measures the average end-to-end time from start to finish of a thread (transaction, query-response, etc). This is typically very important to user acceptability, since users are unhappy waiting long times for completion of a transaction. On the other hand, throughput, measured as thread completions per unit time, is important to a service provider who wants to handle many clients at once. If thread executions require time spent in waiting for input/output, for example, then one can have high throughput even with high latency, since many clients can wait concurrently, with only a few ready to use the CPU at any instant of time.

A performance understanding tool must take these differing needs into account. CPPROFJ does this by providing two different types of views on performance. A *Server View* type profile reports solely on CPU time spent in all threads. This allows the user to see which call paths are accounting for significant CPU usage. This enables optimizing for throughput, since by minimizing CPU usage we can handle more thread instances per unit time.

A *Thread View* type profile, by contrast, accounts for *all* time spent in instances of a single thread class and includes time spent when the thread is not executing. For this, we must define several types of time:

- `:WAIT` time is spent not executing, typically waiting for input (but not for a monitor);
- `:MONITOR` time is spent waiting for an object monitor to be released;
- `:LOAD` is time spent when the thread *could* run but is not because a thread of the same priority is running;
- `:PRIO` is time spent when the thread *could* run but is not because a higher priority thread is running;
- `:RUN` time is time spent when the thread is actually executing (this is the only type of time reported in a Server View profile).

The last three time types are the *runnable* types, while the first two are the *waiting* types. Depending on limitations of the particular Java Virtual Machine (JVM), it may not be possible to get the necessary information to report all of these types as distinct. This is a limitation of the Java JPDA API used by CPPROFJ to get the information; one hopes that in the future more JVMs will come to implement the full JPDA capabilities, of course.

Note that it makes sense to (immediate downward) refine some of the time types. Such a refinement of `:PRIO` or `:LOAD` tells us the thread classes for which the current thread is waiting. `:MONITOR` can be refined into the classes of threads holding the monitor.

By way of illustration, the Email System case study (see Section 6) shows the usefulness of the Thread View and typed time. The client piece is a load tester for a novel email system I developed independently. On first run, each client (which generated and transmitted email messages as fast as possible one after another) was much slower than expected, due (it turns out) to high latency in the threads which actually sent the messages. Each client has a thread that creates and enqueues messages and another thread that actually sends them to the server over the network. Using CPPROFJ I looked at a Thread View profile for the `SMTP_sender` threads. It looked in part like this:

```
1.000 SMTP_sender.run
0.864   :PRIO
...
```

This immediately showed that the threads' latency was due in large part to the sender threads being preempted by higher priority threads. An immediate downward refinement shows more detail:

```
1.000 SMTP_sender.run
0.864   :PRIO
0.607     eventQueueServer
0.131      j.lang.ref.Reference$Handler
0.124      j.lang.ref.Finalizer$Thread
```

This shows that the `eventQueueServer`, which generates the messages, and two garbage collection related threads were getting priority. This points to the need to increase the priority of the sender threads, since they do little computation anyway.

5. CPPROFJ and Its Implementation

The tool is built in two parts. The data collection part, the SAMPLER, runs while the target program is running (typically on a separate machine on the network). It occasionally interrupts the running program and records the runtime stacks of all threads currently in the system, as well as monitor information, if that is available, thread priorities, etc. It writes this data to an intermediate file, the *SampleData* file; it also writes a *Strings* file mapping strings (e.g. method identifiers, object references) to integer codes used in the SampleData file to represent them.

The second part, known as the *Analyzer*, is what the user runs in order to view the profile data. The analyzer operates on the SampleData and Strings files and presents the user with an initial window giving summary information about the thread classes that were observed. The user then has the option of creating one or more Thread View windows (one for each thread class) and a Server View window. Each profile window starts by presenting a single top level node and its cost, which is either 1.000 (for all Thread View windows, since costs in Thread View represent fractions of the average thread lifetime and the top level node of this view must be the entire lifetime), or else a number between 0.000 and 1.000 for the Server View. (A number less than 1.000 in the server view indicates the target program was idle for some of the time.) In each profile window, the user may click on a profile entry to highlight it and then click on one of the three profile types to compute and merge in the results of a refinement profile of the highlighted entry.

Profile entries are presented in a hierarchical fashion, with siblings sorted in decreasing order of cost. Thus, since multiple levels of the hierarchy may be present in the same profile, the overall ordering is not sorted by cost. (Due to the verbosity of call paths, the tree representation was chosen as by far the most compact display.) To help the user quickly visually locate call paths having significant costs, I have included a color icon with each line of the profile display, with the warmer colors indicating higher costs. The user can limit the display to only entries having costs above a threshold by adjusting a simple slider.

5.1. Data Collection by Stack Sampling

The key novelty in the present work with respect to sampling-based profiling is the insight that it can now be done in a portable way within Java itself, using the Java Platform Debugger Architecture (JPDA) API. Previously, sampling based profilers had to be implemented for each combination of hardware architecture, operating system, and language. JPDA, however, gives the Java programmer the hooks necessary to get this information without requiring special purpose code in the JVM.

The basic idea of sampling was originally proposed by Knuth[10]: just randomly stop the running program, see what it is doing, and then let it continue. Do this over and over to build up a statistical picture of the program's behavior over the course of the run. Specifically, CPPROFJ's Sampler uses JPDA to halt the target VM, asks it for a list of the currently running threads, records the method running in each stack frame for the runtime stack of each thread, as well as other information when available, and then continues the VM. This can be done either strictly periodically or at a randomly varied interval. The latter can be better statistically (leads to more representative sampling).

Of course, completely halting a running VM is rather drastic, so to keep overhead (the degree of disturbance to the program's behavior caused by the sampling process itself) down, we sample at a very slow rate compared to program execution speed. For the case studies I have used either once every 10 seconds or once every 5 seconds. For long running applications (such as servers), this is perfectly adequate to get a good picture of program performance. Of course, one must be a bit careful in interpreting profile results when they represent very few sample hits. For example, call paths that had 1 sample hit can have costs that vary quite a bit from the reported cost, due to simple random chance. In a previous paper[5], I prove that a reasonable heuristic (the "ten samples heuristic") is to ignore costs (or at least be suspicious of them) that represent 9 or fewer hits. By default, the slider is set to the 10-hit level when the window first appears, but the user may adjust it as desired.

5.2. Profile Computation

This paper lacks the space necessary to do an adequate job of explaining all the algorithms necessary to computing costs and refinement profiles. A forthcoming longer version of this paper will describe them in detail. The previously published technique[5] must be extended to handle multiple threads, aspects, multiple views, typed time, and upward split refinements. In this subsection I will give a brief sketch of the algorithm for computing the cost of a call path. Refinement computation, which is based upon the same concepts but made more complex by the need to efficiently compute many costs in one pass through the data, will be explained in detail in the long version of this paper.

Each time the Sampler stops the executing JVM, it collects a *sample*. A sample consists of a stack trace for each existing thread, plus ancillary information per thread, including its run state, held and awaited monitors, and priority. The Analyzer reads the SampleData and Strings files and builds a datastructure representing the sample information. Since we only need to collect at most a thousand samples (and typically more like a 100-200) to get an adequate picture of program performance for practical use, this datastructure easily fits into modern memory sizes, which are roughly 32 times larger than when the original call path profiler work was done.

A general utility operation in the implementation is MAPCP(C, f). This method iterates over all thread instances in all samples and calls the user-supplied function parameter f on each thread instance that satisfies the call path. MAPCP recognizes those thread instances by recognizing each of the calls of C in sequence as it traverses the stack trace. pseudo-methods in the call path are recognized by matching its defining pattern to the true method names in the stack trace.

To compute the cost of a call path, we call MAPCP, giving it the call path as argument and a function which

Thread View:

	t runnable		t not runnable	
M	$h > 0$	$h = 0$	$h > 0$	$h = 0$
`:RUN`	0	$1/e$	0	0
`:LOAD`	0	$(e-1)/e$	0	0
`:PRIO`	1	0	0	0
all other	1	1	1	1

Server View:

	t runnable		t not runnable	
M	$h > 0$	$h = 0$	$h > 0$	$h = 0$
`:RUN`	0	$1/e$	0	0
`:LOAD`	0	0	0	0
`:PRIO`	0	0	0	0
`:MONITOR`	0	0	0	0
`:WAIT`	0	0	0	0
all other	0	$1/e$	0	0

Table 1. Sample hits for thread instance t

simply counts the number of recognized thread instances. Once MAPCP completes, we divide that number by the total number of samples. For Server View costs, we divide by all samples taken during the Sampler run; for Thread View costs for thread class T, we divide by the total number of thread instances of class T in all samples.

The above is correct for call paths not ending in one of the time type pseudo-methods: :RUN, :LOAD, :PRIO, :MONITOR, :WAIT. For these, we must have special rules for how many samples are counted at each thread instance.

In other words, let the call path whose cost we seek (C) end in the method (or pseudo-method) M. Suppose MAPCP has recognized the current thread instance t as satisfying C. Let h denote the number of thread instances in the current sample that are both in the *runnable* state and also of higher priority than t. Let e denote the number of runnable thread instances (including t) that are of equal priority with t. Then Table 1 shows the number of sample hits to attribute to t under different conditions.

As shown, in Server View profiles, only runnable threads are counted for call paths ending in the `:RUN` pseudo-method. Moreover, since equal priority threads share the processor equally (presumably), each such instance only gets $1/e$ of a hit. The same is true for other (higher level) call paths, since their runnability implies they are getting $1/e$ of the processor as well. All other Server View entries are 0, reflecting various conditions when the thread instance is not running or else the call path is denoting time spent awaiting other threads.

For Thread View, since we are really measuring clock time (not only CPU time), call paths representing wait time get nonzero hit counts. `:PRIO` call paths get 1 hit only when t is runnable but not running due to higher priority threads. `:LOAD` call paths get $(e-1)/e$ hits, because only $1/e$ of the time in such situations will t actually be running.

The notion of attributing fractional sample hits to concurrent threads was used in the Quartz profiling system[1]. However, the integration of that approach here and extension to handle multiple time views and thread priorities is new with this work. CPPROFJ uses an exact rational arithmetic data type to record and combine fractional hits, so no precision is lost in the summing.

6. Case Studies

This section describes three case studies of applying CPPROFJ to realistic Java applications. In each, I describe optimization opportunities discoverd using CPPROFJ, as well as measurements of overhead and performance improvements.

6.1. Email System Case Studies

The first two case studies are taken from a novel email system. I profiled both a message generation client program and the email server itself. The purpose of the client program is to generate and send messages to the server and retrieve them from it in order to supply a load test for the server. The server is a more or less typical email server that accepts messages from the Internet via the SMTP protocol and serves them back to clients using an encrypted variant of the POP3 protocol. The primary goal of the server is high throughput, while the primary goal of the client is low latency so that it can send and receive as many messages as it can as quickly as possible. Both client and server were written independently as part of another project and then I examined them using CPPROFJ as part of this work, looking for performance problems. The server and client together are approximately 24000 lines of Java code.

The Client. I ran the system with one client feeding one server in various conditions, as shown in Table 2. I used CPPROFJ to find performance bugs and then optimized the most signficant ones. These bugs were described in previous sections of this paper, so I will not repeat them here. As the table shows, however, the improvements were significant when run on both JVMs. Throughput (messages/second) increased from 0.6 to 3.0 on the interpreted (JPDA) JVM, and the same code increased from 10.4 to 12.5 when run in the Hotspot (JIT compiled) JVM. The improvement was not as dramatic in the production JVM, because (in part) the interpreted JVM was slow enough that the jobs became compute bound. Thus, the performance bugs affecting CPU usage were exacerbated in that JVM and hence fixing them had a larger effect.

JVM mode	original	optimized
No JPDA	10.4	12.5
JPDA no sampling	0.5	3.1
JPDA 10sec sampling	0.6	3.0

Table 2. Email client case study throughputs

JVM mode	original	optimized
No JPDA	13.7	14.1
JPDA no sampling	12.8	13.7
JPDA 10sec sampling	12.6	12.8

Table 3. Email server case study throughputs

Note that the overhead due to sampling was small ($\leq$ 3.3%) both for the original and the optimized code bases. (I believe the "negative overhead" in the unoptimized case was due to noise load on the client machine.)

The Server. For the server case study, I ran two faster clients sending messages in order to more fully load the server. I then collected a profile on it using 10 second sampling. One optimization opportunity discovered significantly contributed to thread latency: the SMTP service threads were monitor-waiting when they had to add reference counts to messages (or subtract them from) the message queue's data structure. These operations were synchronized, because the message queues are shared among many threads. However, looking at the code, too much work is done within the critical section; the record could be located first, prior to acquiring the monitor, and only the increment or decrement operation could be within the critical section. This problem was found from the following Thread View profile (fragment) for the SMTPServiceThread class.

```
1.00 SMTPServiceThread
0.52  SMTPServiceThread.finalizeMsg
0.21   :MONITOR
0.19     SMTPServiceThread
0.14      SMTPServiceThread.finalizeMsg
```

As shown in Table 3, overhead due to sampling was from 2 to 9 %. The optimizations found resulted in (modest) speedup both in the JPDA-mode and production JVMs.

6.2. BOB Case Study

BOB (Business Object Benchmark) is a Java benchmark based upon the Transaction Processing Performance Council's[14] TPC Benchmark C (TPC-C). It is described in a book on enterprise Java performance[7] and the code is provided on an associated web site. It is designed to be an object oriented version of TPC-C for measuring Java systems. As a profiler test, I ran it in a lower scale mode (full test, 10 warehouses, 10 terminals per warehouse, 100 threads, 1% population level). It models order fulfillment and payment for goods moving through warehouses.

JVM mode	ramp-up (m)	test (m)	score (tps)
No JPDA	26.5	20.5	216
JPDA no sampling	21.9	20.0	102
JPDA 5sec sampling	21.6	21.0	100

Table 4. BOB case study measurements

When run, the benchmark first enters a (roughly) 20 minute "ramp-up" phase and then executes a 20 minute "test" phase; it then produces a score representing transaction throughput. As in the email studies, I ran it in three different conditions. Table 4 shows the results, for overhead comparison purposes. As we can see, sampling only reduced throughput by 2% compared to the same JVM without sampling. Apparently, JPDA mode slows ramp-up time by about 7 or 8 %. The test phase was slowed by sampling about 5% as well, even though the throughput was not reduced by that much. (The mysterious increase in ramp-up time for the production JVM may be worth investigating.)

I then examined the profile collected using Analyzer to search for performance bottlenecks in the BOB code. The Server View shows an interesting fact:

```
0.418 *
0.248   ..j.util.Date.getField
```

which shows that over half of the CPU time actually used is spent getting various parts of the current date and time. Examining a Thread View profile of the `CompanyRunnable` thread class (the only one of interest), we see that the average thread spends 83.9% of its latency time waiting for some other `CompanyRunnable` thread to get Date fields. Note the interleaved use of immediate and extended refinements to get to this information via five mouse clicks.

```
1.000 CompanyRunnable
1.000   CompanyRunnable.run
0.911     TransactionManager.go
0.895       ..:MONITOR
0.894         CompanyRunnable
0.839           ..j.util.Date.getField
```

If this were a real application and we were concerned about thread latency, then we could likely reduce the work done in constantly getting the date fields and certainly get rid of monitor waiting for this operation.

7. Discussion

Related Work. Profiling is, of course, an old idea, and there are many flavors of profiler. *Body profilers* such as Unix's `prof` profiler, report costs incurred executing in the body of each computation unit only. Time in call descendants is not included. This measurement is obtainable in CPPROFJ as an upward split refinement of the call path `(* ..:RUN)`. In fact, CPPROFJ allows related profiles such as upward splits of `(* ..:PRIO)` and `(* ..:MONITOR)`. *Method (function) profilers* also report time spent in each computation unit, but *include* time in all call descendants. Many tools produce method profiles, including Sitraka's JProbe[13]. A method profile can be produced easily in CPPROFJ as the extended downward refinement of `(*)`. This can be done in either type of view. *Call graph profilers*[4, 13] report costs incurred in all 1-level call contexts; that is, they report costs for all length 2 call paths. This information can be obtained from CPPROFJ, if desired, but the more useful refinement-based approach in CPPROFJ ignores the large majority of length 2 call paths (which are insignificant anyway) and instead focuses on signficant length 2 (and longer) call paths.

The common drawback of all these profilers is that while providing copious low level information, they do not relate costs to design decisions *within arbitrary call context*. As the char-to-byte example in Section 2 showed, call contexts as deep as six and more are common, particularly when software is constructed in a layered, object-oriented fashion where classes inherit behavior from reusable ancestor classes. This difference between CPPROFJ and lower level profilers is fundamental: one can easily show by (common) example that the information in a call path profile cannot in principle be inferred from the information presented even in body, function, and call graph profiles together.

Stack tree profilers[12, 3, 11], on the other hand, do collect and present information on arbitrarily deep call stacks. However, they present this information as a single tree rooted at the (conceptual) `main` routine. Thus, resource usage that occurs in many widely-spaced calls distributed over the execution (such as char-to-byte conversion routines, `msg_queue` methods, and database functions, for example) will have their costs obscured by the need to manually add up many small contributions. None will stick out of the display. CPPROFJ, on the other hand, is designed to focus attention by aggregating many aspects together, such as typed time and aspect pseudo-methods. Another important difference was alluded to in Section 2: because they do not aggregate information on recursion as does a call path, they do not correctly report on costs of calls from recursive methods to other methods, *when the goal is to gauge the likely impact of optimizations.* This is because the cost of `A B C` will be reported separately from the cost of `A B D E B C`, due to the tree structure of the display.

The present work is based upon previous work on call path profiling[5, 6], but there are some notable differences. Previous work explored profiling *arbitrary monotonic resources*, of which time is only one example. Others include space usage, cache misses, page faults, and system calls. Any strictly increasing quantity could be used. The present work only works for the typed time described above, because that is all I know how to obtain data for in a portable way. It is always possible to implement specialized data measurement infrastructure if one controls the JVM to be used, which is the approach in many tools. The Analyzer piece of CPPROFJ can be easily adapted to any such resource for which the SampleData and Strings files can be obtained, as long as those input files do something reasonable in associating resource consumption with the typed time pseudo-methods. As stated earlier, the upward split refinement profile is new in this work, as are the distinction between Server and Thread Views, and support for typed time within Thread View. Previous work had a notion of "grouping," which was a less expressive precursor to the present notion of aspect. The useful capability of *pruning* arbitrary time from the profile (described by boolean expressions over call paths), which was present in the earlier work, has yet to be reimplemented in CPPROFJ.

Aspect oriented programming[2] is the study of how to conceptualize and implement software in a way that separates concerns as much as possible (for ease of comprehension, maintenance, and reuse), without, hopefully, sacrificing too much in performance. There are a number of language extensions that allow explicit aspect-oriented programming in Java. One of the better known is AspectJ[9]. It provides language constructs for defining cross-cutting aspects of the code; when control flow reaches an aspect boundary (such as a call to a method belonging to an aspect definition), user-specified code can be called. Typical uses for "development" aspects (those used during code development) might be to implement tracing, debugging, testing, or instrumentation-based profiling tools. "Production" aspects (those used in deployed code) can do things like implement a display tracking changes in a model, such as updating the graphical display when a point moves in a 3-D model. AspectJ allows defining aspects by name (i.e. patterns matching against method names) and property-based, where the pattern can match against signature elements or can refer to properties of the control flow of the program.

CPPROFJ's implementation of aspects is still in its initial stages. Currently, we only allow simple wildcarded name-plus-argument-types specifications (e.g., no matching on the return type). However, the AspectJ notion of "controlflowbelow" as used in defining an aspect is captured naturally in call path profiles, since the costs reported are always covering all time when control flow satisfies the call path, including time in call descendants. As described earlier, CPPROFJ can report costs incurred by cross-cutting design decisions represented by aspects. If the code is implemented in AspectJ (or some other aspect-oriented language extension of Java), then it should be relatively easy to change the design decision if an optimization is discovered. If the code is not written in an aspect oriented way, a CPPROFJ user can still define the aspects himself and locate optimization opportunities using CPPROFJ. How easy it is to actually carry out the optimizations so discovered is, of course, dictated by the actual code structure.

Limitations and Future Work. Unfortunately, not all modern JVMs work with the JPDA API equally well. Sun's Hotspot JVM, for example, must run in interpreted mode, without the just-in-time (JIT) compiler. Thus, the application may run much slower in this mode than it would in production use. While this is clearly undesirable and can obscure some performance issues, most performance bottlenecks present in the code should manifest when run in either JVM. Of course, it may be necessary to scale down the test input in order to profile the system in a comparable load regime. This has not been a difficulty in the case studies. And, as they show, it is possible to find optimization opportunities using CPPROFJ that result (once the code is optimized) in performance improvements in the system as run under the production JVM. I hope, as the use of JPDA for sampling-based profiling becomes well known, that JVM vendors will build in support for it into the production/compiled modes of their JVMs. Perhaps even better would be a special API to support Sampling based profiling, which does not need the full capabilities of JPDA.

The sampling process itself introduces some overhead, causing deviation from the program's behavior by occasionally halting the JVM. By reducing the (average) frequency of Sampler, we can reduce this overhead to acceptable levels. JPDA, being primarily a debugger interface, is not optimal for sampling. Getting stack trace and other thread information requires several individual remote procedure calls across the network *for each stack frame*. This takes longer than it would if the API had commands that could stream this information in response to a single query. Hopefully, this observation will inform future API designs.

As discussed above, previous call path profilers were capable of profiling other resources than time. Once there is a portable way to do this in Java, or if someone designs a special JVM to capture this information, CPPROFJ can be easily adapted to handle it. See [6] for various techniques.

8. Conclusion

The enhanced approach to call path profiling in this paper supports performance understanding of modern multi-threaded client and server applications by enabling a user to directly associate costs with optimizable design decisions. Call context enabled specializations, cross-cutting aspect implementations, and thread dependencies are three common categories of optimization opportunity to which costs are directly associated by the techniques described here: the various refinement profile types (including upward split refinements), two views of performance (Server and Thread), typed time, and support for aspects. CPPROFJ is an all-Java tool which exploits the recent development of the JPDA API to achieve accurate and portable sampling-based profiling for Java. However, the ideas are readily transferrable to other languages and platforms; in fact, the Analyzer portion of CPPROFJ can be used as is, assuming a Sampler can be written for the other platform that produces SampleData and Strings files in the defined format. The examples and case studies in this paper provide support for the claim that this approach is a useful next step toward high-level automated program performance understanding.

References

[1] T. Anderson and E. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proc. of ACM SIGMETRICS 1990 Conf. on Measurement and Modeling of Computer Systems*, pages 115–125. ACM, 1990.

[2] Aspect-oriented software development (web site). `http://www.aosd.net`.

[3] Franz Inc. *Allegro COMPOSER User Guide, version 1.0*, 1990.

[4] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, June 1982.

[5] R. J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6), June 1995.

[6] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in unix. In *Proc. 1993 USENIX Summer Technical Conf.* USENIX Association, 1993.

[7] S. L. Halter and S. J. Munroe. *Enterprise Java Performance*. Prentice Hall, 2001. Chapter 8.

[8] Java platform debugger architecture overview (web page). `http://java.sun.com/j2se/1.3/docs/guide/jpda/jpda.html`.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Comm. ACM*, 44(10):59–65, October 2001.

[10] D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1:105–133, 1971.

[11] J. Larus and T. Ball. Efficient path profiling. In *Proc. MICRO-29*. IEEE, 1996.

[12] Pure Software Inc. *Quantify User's Guide*, 1993.

[13] Sitraka jprobe (web page). `http://www.sitraka.com/software/jprobe/`.

[14] Transaction processing performance council (web site). `http://www.tpc.org/`.