Finding and Removing Performance Bottlenecks in Large Systems

Glenn Ammons[1], Jong-Deok Choi[2], Manish Gupta[2], and Nikhil Swamy[3]

[1] IBM T. J. Watson Research Center,
Hawthorne, New York, USA,
ammons@us.ibm.com

[2] IBM T. J. Watson Research Center,
Yorktown Heights, New York, USA,
{jdchoi,mgupta}@us.ibm.com

[3] Department of Computer Science, University of Maryland,
College Park, Maryland, USA,
nswamy@cs.umd.edu

Abstract. Software systems obey the 80/20 rule: aggressively optimizing a vital few execution paths yields large speedups. However, finding the vital few paths can be difficult, especially for large systems like web applications. This paper describes a novel approach to finding bottlenecks in such systems, given (possibly very large) profiles of system executions. In the approach, for each kind of profile (for example, call-tree profiles), a tool developer implements a simple profile interface that exposes a small set of primitives for selecting summaries of profile measurements and querying how summaries overlap. Next, an analyst uses a search tool, which is written to the profile interface and thus independent of the kind of profile, to find bottlenecks.

Our search tool (Bottlenecks) manages the bookkeeping of the search for bottlenecks and provides heuristics that automatically suggest likely bottlenecks. In one case study, after using Bottlenecks for half an hour, one of the authors found 14 bottlenecks in IBM's WebSphere Application Server. By optimizing some of these bottlenecks, we obtained a throughput improvement of 23% on the Trade3 benchmark. The optimizations include novel optimizations of J2EE and Java security, which exploit the high temporal and spatial redundancy of security checks.

1 Introduction

J. M. Juran's Pareto principle [19, 16] (also known as the 80/20 rule) admonishes, "Concentrate on the vital few, not the trivial many". For software systems, the Pareto principle says that aggressively optimizing a few execution paths yields large speedups. The principle holds even for large systems like web applications, for which finding the "vital few" execution paths is especially difficult. Consequently, finding bottlenecks in such systems has been the focus of much previous work [27, 22, 1].

This paper describes a novel approach to finding bottlenecks, given (possibly extremely large) profiles of one or more executions of a system. In our approach, for each kind of profile (for example, call-tree profiles), one implements a simple interface that supports searching for bottlenecks. Next, an analyst uses a search tool, which is independent of the kind of profile, to find bottlenecks. The tool manages the bookkeeping of the search and provides heuristics that automatically suggest likely bottlenecks.

Our search tool is called Bottlenecks. In one case study, after using Bottlenecks for half an hour, one of the authors found 14 bottlenecks in IBM's WebSphere Application Server (WAS) [30]; optimizing six of these bottlenecks yielded a 23% improvement in throughput on the Trade3 benchmark [29]. Although the author was already familiar with several of these bottlenecks from spending many days inspecting call-tree and call-graph profiles without the aid of Bottlenecks, by using Bottlenecks, the author quickly verified the familiar bottlenecks and found new bottlenecks. Moreover, one of the new bottlenecks suggested a new optimization of Java 2 security.

```
Cost: 25% of total
Context:
  CCommandImpl.execute
  TCommandImpl.setOutputProperties
  AccessController.doPrivileged
  TCommandImpl$1.run
  Class.checkMemberAccess
  SecurityManager.checkMemberAccess
  SecurityManager.getClassContext
```

```
Cost: 13% of total
Context:
  EJSSecurityCllbrtr.preInvoke
  AccessController.doPrivileged
  EJSSecurityCollaborator$1.run
  CurrentImpl.get_credentials
  SecurityManager.checkPermission
  SecurityManager.checkPermission
  AccessController.checkPermission
  AccessController.getStackACC
```

Fig. 1. Two WAS bottlenecks, which we found with Bottlenecks. Each bottleneck summarizes execution-cost measurements by the calling context in which they were taken. Method and class names have been shortened to fit on a line.

Finding bottlenecks in large systems by hand is hard because of two related problems: choosing the best way to summarize execution-cost measurements, and keeping track of overlap among bottlenecks. The bottlenecks in Figure 1 illustrate the problems. These bottlenecks are real bottlenecks in WAS, which we found with Bottlenecks. Each bottleneck lists a sequence of calls, and summarizes execution-cost measurements taken in calling contexts [4] that contain the sequence. In this example, execution-cost measurements record the instruction-count overhead of enabling security in WAS; for example, if the application executes one billion more instructions when security is enabled than it does when security is disabled, then the bottlenecks in the figure account for 250 million and 130 million of those instructions, respectively.

[4] The calling context of a measurement records the name of the active method, for each activation record on the call-stack when the measurement is taken.

There are two reasons to summarize measurements: efficiency and understandability. For efficiency, profilers summarize measurements on-line, as the system runs. For example, flat profiles keep one summary per basic block, control-flow edge, or method; call-tree or calling-context-tree profiles [2] keep one summary per calling context; call-graph profiles keep one summary per call-graph edge; and Ball-Larus path profiles [4] keep one summary per intraprocedural, acyclic path.

Understandability of measurements is also crucial because human analysts cannot comprehend large numbers of measurements without summaries. In fact, profilers are usually distributed with report-generation tools that can reduce profiles to flat profiles, no matter how the profiler summarizes measurements internally.

The problem with summarization is that no summarization scheme suffices to find all bottlenecks. For example, neither a call-tree nor a call-graph profile is adequate for finding the bottlenecks in Figure 1. A call-tree profile is inadequate because both bottlenecks occur in many different calling contexts, and so in many different locations in the tree; a call-graph profile is inadequate because it has one summary for `doPrivileged`, while finding the bottlenecks requires a summary for `doPrivileged` when called by `setOutputProperties` and another summary for `doPrivileged` when called by `preInvoke`.

Two bottlenecks *overlap* if both summarize measurements of some common cost. Keeping track of overlap among bottlenecks is the second problem in finding bottlenecks.

For example, the bottlenecks in Figure 1 have no overlap with one another, because no execution-cost measurements occurred in a calling context that contained both the call sequence on the left and the call sequence on the right. On the other hand, both bottlenecks overlap with `doPrivileged`, because a calling context that contains either sequence in the figure also contains a call of `doPrivileged`.

Computing overlap manually is difficult in large profiles, but without computing overlap it is impossible to estimate the potential speedup of optimizing a set of bottlenecks. Because of overlap, while optimizing bottlenecks separately may yield performance improvements in each case, optimizing bottlenecks together might not yield the sum of their separate improvements.

Because estimating speedup requires computing overlap, overlap must be tracked during the search for bottlenecks. Otherwise, the search could return redundant bottlenecks, which are not worth optimizing.

Our method for finding bottlenecks addresses both the summarization problem and the overlap problem. In our method, a profile interface defines a small set of primitives that support constructing summaries of execution-cost measurements and computing the overlap of summaries. The interface associates summaries with execution paths (for example, call sequences), instead of summarizing according to a fixed summarization scheme like call-graphs or call-trees. Of course, any given profiler *will* use some fixed summarization scheme; however, the interface presents a flexible mechanism for automatic tools or human ana-

lysts to summarize further. And, because the interface makes overlap explicit, speedup can be estimated directly from any collection of summaries.

Because the profile interface can be implemented for any profile that associates execution-cost measurements with paths, it isolates analysis tools from details of the profiles. In fact, by including two implementations of the profile interface, Bottlenecks supports analyzing call-tree profiles in two very different ways.

The first implementation supports finding expensive call sequences by analyzing call-tree profiles. This implementation provides the full precision of the call-tree profile only where it is needed; where precision is not needed, measurements are summarized just as fully as they are in a flat profile, without losing the ability to estimate speedup.

The second implementation is comparative: it supports finding call sequences that are significantly more expensive in one call-tree profile than in another call-tree profile. For example, if a system is slower in one configuration than another, the cause can be found by comparing a profile of the slow configuration with a profile of the fast configuration.

No matter which implementation is in use, Bottlenecks presents the same user interface to the performance analyst. This interface manages the bookkeeping of the search and provides simple heuristics that automatically suggest likely bottlenecks.

Contributions This paper makes the following contributions:

- A novel method for finding performance bottlenecks, given program execution profiles.
- Two instances of the method, both of which we have implemented within Bottlenecks. One instance supports finding expensive call sequences in call-tree profiles, while the second supports finding call sequences that are significantly more expensive in one call-tree profile than in another call-tree profile.
- A report on the bottlenecks we found with Bottlenecks and optimizations that remove them. Among the latter are novel optimizations that remove much of the overhead of enabling the security features of WAS. These optimizations exploit two properties: the same security checks are made several times in close succession (that is, they have high *temporal redundancy*) and the same checks are made repeatedly on the same code paths (that is, they have high *spatial redundancy*).

2 Finding Bottlenecks

We started the Bottlenecks project after a painful experience in analyzing large profiles of WAS with traditional tools. Based on that experience, and on the observations in Section 1, we had three goals for a better method:

Adequate power The method must allow an analyst (human or machine) to vary how profiles are summarized and to account accurately for overlap among summaries.

Profile independence The method must place as few requirements on input profiles as possible.

Extensibility Because finding bottlenecks is not a well-understood problem, it must be easy to improve the method iteratively by inventing and validating new interfaces and algorithms to support the search for bottlenecks.

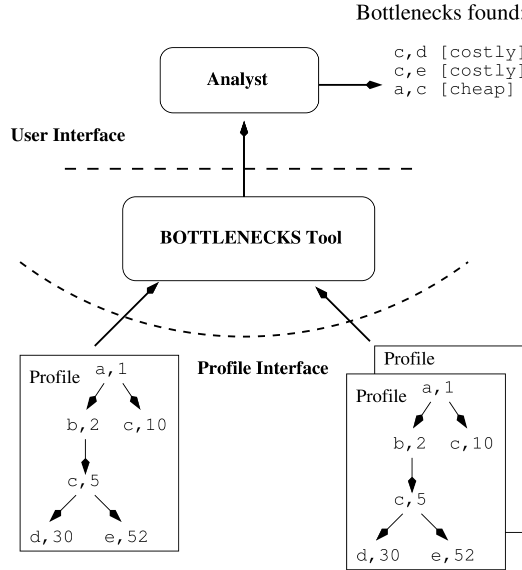

Fig. 2. The architecture of Bottlenecks.

Figure 2 is an overview of our Bottlenecks tool, which satisfies these goals. Bottlenecks has two parts: the *profile interface* and the *user interface*.

The profile interface defines an abstraction with adequate power for navigating and analyzing profiles. Specifically, the interface provides primitives for constructing various summaries (with associated execution paths) and for computing the overlap among summaries. The interface is profile independent, because these primitives can be implemented for any profile that associates a metric with paths; as the figure shows, Bottlenecks has one implementation for call-tree profiles and another for comparing two call-tree profiles.

The user interface is the abstraction that a human analyst sees. In its simplest usage mode, the user interface allows navigating a profile by inspecting summaries of longer and shorter execution paths. At this level, the tool helps

the analyst by managing tedious bookkeeping such as remembering which summaries have been inspected. In addition, the user interface is designed to be extended with algorithms that suggest starting places for the analyst's search or that automate parts of the search.

The outline of the rest of this section is as follows. Section 2.1 discusses the profile interface. Section 2.2 explains how we implemented the interface for call-tree profiles and for comparing call-tree profiles. Finally, Section 2.3 describes the user interface and gives an example of its use.

2.1 The Profile Interface

The profile interface is a simple abstraction that supports constructing summaries and computing overlap among summaries. Specifically, the interface has functions to generate an initial set of summaries, given a profile; to query summaries; to construct new summaries from old summaries; and to compute overlap among summaries.

```
module type PROFILE_INTERFACE =
sig
  type t
  type profile_t
  val path_of : t -> string list
  val initial_summaries : profile_t -> t list  (* Construction. *)
  val top_extensions : t -> string list
  val bottom_extensions : t -> string list
  val extend_top : t -> string -> t
  val extend_bottom : t -> string -> t
  val trim_top : t -> t option
  val trim_bottom : t -> t option
  val base_of : t -> int64                     (* Metrics. *)
  val cum_of : t -> int64
  val total_base_of : t list -> int64
  val total_cum_of : t list -> int64
end
```

Fig. 3. The profile interface: a standard interface to profiles that associate a metric with paths.

Figure 3 lists Objective Caml [23] code for the profile interface. The type `t` is the type of summaries; the type `profile_t` is the type of the underlying profile. The code in the figure lists interface functions with their types but without their definitions—for example, the function `initial_summaries` accepts a profile as its sole argument and returns a list of summaries.

A central principle of the interface is that each summary corresponds to an execution path, which is simply a sequence that identifies a set of execution-cost

measurements (for example, a call sequence). The function `path_of` returns a summary's execution path, which is assumed to be representable as a list of strings (for example, a list of method names).

The profile interface has seven functions for constructing summaries. Given a profile, the function `initial_summaries` returns a set of basic summaries, which, in a typical implementation, correspond to execution paths of length 1.

The other six functions enable constructing summaries that correspond to longer execution paths. These include functions to query the profile for the list of possible additions to the front or back of a summary's execution path (`top_extensions` and `bottom_extensions`), to create a new summary by adding a string at the front or back (`extend_top` and `extend_bottom`), and to create a new summary by trimming one string from the front or back (`trim_top` and `trim_bottom`).

The profile interface supports two metrics for each summary. We assume that the profile associates (implicitly or explicitly) a metric with each execution path. The function `base_of` returns this metric, which we call the *base* of a summary. Some profiles also have a concept of a cumulative metric; for example, the cumulative cost of a node in a call-tree profile is the cost of the node itself plus the cost of all its descendants. For such profiles, the function `cum_of` returns the cumulative metric of a summary's execution path, which we call the *cum* of the summary.

Finally, `total_base_of` and `total_cum_of` return the total base and cum of a list of summaries. The intent is that these functions should account for any overlap among the summaries: even if an execution-cost measurement belongs to more than one summary, it should be counted only once.

Other useful functions about overlap can be defined in terms of `total_base_of` and `total_cum_of`.[5] For example, this function returns the cum overlap of a summary s with a list of summaries S (in Objective Caml, the `::` operator conses a value onto the head of a list):

```
let cum_ol s S =
  (cum_of s) + (total_cum_of S) - (total_cum_of (s :: S))
```

The user interface of BOTTLENECKS assumes only that `base_of`, `cum_of`, `total_base_of`, and `total_cum_of` are implemented as functions of the right type. There are no other assumptions. In fact, comparative profiles violate many "common sense" assumptions (see Section 2.2): in comparative profiles, both base and cum may be negative and a summary's cum may be smaller than its base. Nonetheless, implementations should not return haphazard values; although BOTTLENECKS does not fix an interpretation of these metrics, a natural interpretation should exist. Section 2.2 explains what these functions compute in our implementations.

[5] Even `cum_of` and `base_of` can be defined in terms of `total_base_of` and `total_cum_of`. For explanatory purposes, we give separate definitions.

2.2 Implementations of the Profile Interface

This section describes two implementations of the profile interface: one for call-tree profiles and another for comparing two call-tree profiles.

Call-tree Profiles To implement the profile interface for call-tree profiles, we must implement the types and functions in Figure 3. The following is a sketch of our implementation, which is both simple and fast enough to navigate call-tree profiles with over a million nodes.

The type `profile_t` is the type of call-tree profiles; a call-tree profile is a tree where

- each node is labeled with a method name and a cost; and
- for each calling context $m_0, \ldots, m_k$ that occurred during program execution, there is exactly one path $n_0, \ldots, n_k$ such that n_0 is the root of the tree and, for each $0 \leq i \leq k$, n_i is labeled with the method name m_i.

Intuitively, a node's cost summarizes all execution-cost measurements that occurred in the node's calling context. The profile that appears (twice) in Figure 2 is a call-tree profile.

Summaries consist of a call sequence and the list of all nodes that *root* the call sequence:

```
type t = { calls : string list ; roots : node list }
```

A node roots a call sequence iff the call sequence labels a path that begins at that node. For example, in Figure 2, the call sequence `[c]` has two roots: namely, the two nodes labeled `c`. By contrast, the only root of `[c;d]` is the left node labeled `c`.

The function `path_of` simply returns the `calls` component of a summary.

The function `initial_summaries` traverses the tree and creates a summary for each length-1 call sequence that occurs in the tree. For example, given the profile in Figure 2, `initial_summaries` creates five summaries: one each for `[a]`, `[b]`, `[c]`, `[d]`, and `[e]`.

Given a summary s, the function `top_extensions` returns all method names m such that m `::` `path_of(`s`)` labels at least one path in the tree; these names are easy to find by inspecting the parents of s`.roots`. For example, if s is the summary for `[c]` in Figure 2, then `top_extensions(`s`)` returns `a` and `b`.

Similarly, `bottom_extensions(`s`)` returns all method names m such that `path_of(`s`)` `@` `[`m`]` has at least one root (in Objective Caml, the `@` operator concatenates two lists); these names are easy to find by inspecting the children of all nodes reachable by following paths labeled s`.calls` from nodes in s`.roots`. For example, if s is the summary for `[c]` in Figure 2, then `bottom_extensions(`s`)` returns `d` and `e`.

Given a summary s and a top extender m of s, `extend_top` returns the summary s' for m `::` `path_of(`s`)`; s'`.roots` can be computed easily from s`.roots` and is never larger than s`.roots`. The definition of `extend_bottom` is similar.

We come now to the definitions of the base and cum metrics. For these, we need some auxiliary definitions (as usual, s is a summary):

paths(s) All paths labeled `s.calls` from nodes in `s.roots`.
along(s) All nodes that are along some path in paths(s).
interior(s) All nodes that are along some path in paths(s) but *not* at the end of any such path.
final(s) All nodes that are at the end of some path in paths(s).
descendants(s) All nodes that are descendants of some node in final(s).

Note that our implementation does not necessarily compute these sets. In particular, descendants(s) can be the entire tree, so computing it for each summary is prohibitively expensive.

Given a summary s, the base of s is given by

$$\texttt{base_of}(s) = \sum_{n \in \text{along}(s)} \text{cost of } n$$

For example, if s is the summary for `[c]` in Figure 2, then `base_of(s)` is 15.

The cum of s also includes the cost of all descendants of s:

$$\texttt{cum_of}(s) = \sum_{n \in \text{along}(s) \cup \text{descendants}(s)} \text{cost of } n$$

For example, if s is the summary for `[c]` in Figure 2, then `cum_of(s)` is 97.

As mentioned above, computing descendants(s) is too expensive. Thus, when it loads a profile, our implementation precomputes a *cum-cost* for each node in the tree: the cum-cost of a node equals its cost plus the cost of its descendants. All cum-costs can be computed in one traversal of the tree. Given cum-costs, `cum_of(s)` can be implemented efficiently by evaluating this formula:

$$\sum_{\substack{n \in \text{interior}(s) \\ n \notin \text{descendants}(s)}} \text{cost of } n \quad + \sum_{\substack{n \in \text{final}(s) \\ n \notin \text{descendants}(s)}} \text{cum-cost of } n$$

This formula can be evaluated quickly because checking for membership of n in descendants(s) can be done in time proportional to the depth of n, by traversing tree edges backwards towards the root.

The reader may be asking why we exclude descendants of s from the sums in the last formula. The reason is that, in the presence of recursion, a node can be in interior(s) or final(s) and also have an ancestor in final(s). If such descendants were not excluded, the sums would count them twice.

To complete the implementation of the profile interface, we must implement `total_base_of` and `total_cum_of`. Intuitively, computing cum and base for a set of summaries S is the same as computing cum and base for a single summary, except that now all paths in S must be taken into account. So, we extend the auxiliary functions to functions over sets of summaries:

paths(S) The union over all $s \in S$ of paths(s).
along(S) All nodes that are along some path in paths(S).
interior(S) All nodes that are along some path in paths(S) but *not* at the end of any such path.
final(S) All nodes that are at the end of some path in paths(S).
descendants(S) All nodes that are descendants of some node in final(S).

Then, the formulas for `total_base_of` and `total_cum_of` are the same as the formulas for `base_of` and `cum_of`, but with s replaced by S. For example, if S consists of the summary for `[a]` and the summary for `[c]` in Figure 2, then `total_base_of(`S`)` is 16 and `total_cum_of(`S`)` is 100.

Comparing Call-tree Profiles It is sometimes useful to compare two profiles. For example, if a system is slower in one configuration than another, the cause can be found by comparing a profile in the slow configuration with a profile in the fast configuration—Section 3 discusses how we applied this technique to reduce the security overhead of WAS. This section describes an implementation of the profile interface that allows comparing two call-tree profiles.

Comparing two call-tree profiles requires deciding how to relate subtrees of the first profile to subtrees of the second profile. Our approach is based on the intuition that analysts are most interested in the cost of paths through programs. Thus, instead of (for example) explicitly constructing a map from subtrees of one profile to subtrees of the other profile, our implementation simply compares the cost of a call sequence in one profile with its cost in the other profile.

An advantage of this approach is that the comparative implementation can reuse most of the code of the implementation for single call-tree profiles. The type of summaries is a slight modification of the type of summaries for single call-tree profiles:

```
type t = { calls : string list ;
           a_roots : node list ;
           b_roots : node list }
```

Instead of one `roots` field, we now have an `a_roots` field that lists nodes in the first profile and a `b_roots` field that lists nodes in the second profile. Thus, a summary s denotes zero or more paths in one tree, and zero or more paths in a second tree.

The function `initial_summaries` traverses both trees and produces a list of paths of length 1, of the form

```
{ calls = [m] ; a_roots = a_ns ; b_roots = b_ns }
```

Here `a_ns` lists all roots of `[`m`]` in the first tree, while `b_ns` lists all roots of `[`m`]` in the second tree. At least one of these lists is not empty.

The other functions are defined in terms of the functions for a single call-tree profile. For example, if `Single.base_of` implements `base_of` for a single call-tree profile, then the comparative `base_of` is defined by

```
let base_of s =
  (Single.base_of ({ calls = s.calls ; roots = s.a_roots}))
  - (Single.base_of ({ calls = s.calls ; roots = s.b_roots}))
```

In general, functions that return numbers are implemented by subtracting the single-tree result for the second profile from the single-tree result for the first profile. Other functions combine the results in other natural ways. For example, `top_extensions` returns the union of the top extensions in the first and the second profile.

Due to the nature of comparing profiles, the comparative implementation lacks several "common sense" properties. For example, if a summary has a higher base in the second profile than it does in the first profile, then the summary has a *negative* base in the comparative profile. For similar reasons, the cum of a summary can be lower than its base. These paradoxes arise because of the nature of comparison; the best that an implementation can do is expose them, so that they can be dealt with at a higher level. In practice, we find that they are not problematic, at least when comparing trees that are more similar than they are different.

2.3 The User Interface

This section describes the user interface of Bottlenecks. This command-line interface provides three kinds of commands: *suggestion commands*, which request summaries at which to start a search for bottlenecks; a *navigation command*, which moves from one summary to another; and a *labeling command*, which assigns labels to summaries. Suggestion and navigation permit the human analyst to find summaries that explain bottlenecks well, without the limitations of fixed summarization schemes like call trees and call graphs; by design, they are also good points at which to introduce automation. The analyst uses labels to mark interesting summaries. Labels are also the mechanism for requesting overlap computations: during navigation, Bottlenecks prints the overlap of each summary with labeled summaries, so that the analyst can avoid investigating redundant summaries.

The rest of this section gives a simplified overview of the user interface and, as an example, demonstrates how to use Bottlenecks to uncover the bottleneck on the left side of Figure 1.

The analyst starts a search by picking a *suggester*:

<`set suggester` *name*> Pick a suggester. A suggester generates an ordered list of starting summaries, based on the profile.

Next, the analyst views the suggestions:

<`suggest`> Print the suggester's starting summaries.

In the future, as we discover better techniques for finding bottlenecks automatically, we will implement them as suggesters. At the moment, Bottlenecks has two simple suggesters:

HighCum The HighCum suggester suggests summaries for paths of length 1 (that is, individual methods), with summaries ranked in descending order by their cum. These summaries are good starting points for a top-down search.

HighBase The HighBase suggester also suggests summaries for paths of length 1, but with summaries ranked in descending order by their base. These summaries are good starting points for a bottom-up search.

Bottlenecks gives a number to every summary it prints. The analyst navigates from summary to summary by selecting them by number:

<`select` n> Select the summary numbered n. The summary (call it s) becomes the *current summary*. Bottlenecks prints details about s:

- If s has been labeled, the labels of s.
- The cum and base metrics of s.
- For each unique label l that the analyst has assigned to one or more summaries, the overlap of s's cum and base metrics with summaries labeled l.
- The execution path associated with s.
- A numbered list of "nearby" summaries, which can be reached by applying summary construction functions (see Figure 3).

Generating the list of nearby summaries is another point at which the user interface can be extended with heuristics. Bottlenecks has two algorithms for producing this list. The first algorithm simply prints all 1-method extensions and trimmings of the current summary.

The second algorithm, called *zooming*, omits low-cost extensions and trimmings and greedily "zooms" through extensions and trimmings that concentrate the cost. The goal is to avoid printing uninteresting summaries: low-cost summaries are uninteresting because the user is unlikely to choose them, while summaries that concentrate the cost are uninteresting because the user is almost certain to choose them. In practice, zooming significantly reduces the time it takes to find useful bottlenecks.

Zooming depends on a user-settable cutoff ratio c, which is a positive real number (the default is 0.95). Zooming uses c both to identify low-cost summaries and to identify summaries that concentrate the cost. The following pseudocode shows how zooming finds nearby top extensions (bottom extensions and trimmings are similar):

```
Routine Zoom(s, c) = ZoomRec(s, c|cum_of(s)|)
Routine ZoomRec(s, C) =
  T  := top extensions of s, in descending order by |cum_of|
  T_z := first N summaries in T, where N > 0 is smallest s.t
          |total_cum_of(T_z)| ≥ C, or ∅ if no such N exists
  If |T_z| = 1 Then Return ZoomRec(first(T_z), C)
  Else Return T_z
```

Sorting T_z by the absolute value of `cum_of` [6] identifies low-cost summaries. The conditional tests for summaries that concentrate the cost: if the cost of a summary is at least C, then the user will almost certainly select it, so the algorithm zooms through it.

For example, suppose that the current summary is for `[cd]` in Figure 2. If zooming were enabled, BOTTLENECKS would zoom to the 2-method top extension `[abcd]` instead of listing the 1-method top extension `[bcd]`.

Finally, BOTTLENECKS provides a labeling command, which the analyst uses to mark interesting summaries:

<`label` *name*> Assign the label *name* to the current summary.

Once labeled, a summary can be inspected later or saved to a file. More importantly, as the analyst searches for bottlenecks, BOTTLENECKS displays the overlap of the current summary with labeled summaries. Accounting for overlap is key to estimating the expected benefit of optimizing a particular bottleneck; therefore, after the first bottleneck has been found, the analyst must take overlap into account when selecting the next summary.

Note that BOTTLENECKS does not interpret labels; labels have meaning for the analyst, not for BOTTLENECKS.

An Example Like other application servers, WAS is slower when its security features are enabled. To find the cause of this slowdown, we ran the Trade3 application server benchmark twice, the first time with security enabled and the second time with security disabled, and compared them with BOTTLENECKS, using the comparative implementation of the profile interface. Figure 1 lists two of the bottlenecks that we found. This section works through a session in which we find the bottleneck on the left side of the figure, using BOTTLENECKS and a bottom-up approach.

The first steps are to choose an appropriate suggester and list the highly ranked suggestions. The HighBase suggester is better for a bottom-up search:

```
set suggester HighBase
suggest
```

BOTTLENECKS prints the summaries with highest base. The highest-ranked summary is for the call sequence

[`SecurityManager.getClassContext`]

which has a base that accounts for 9.85% of the total security overhead—that is, 9.85% of the difference between the cost when security is enabled and the cost when security is disabled. As it happens, this method is never called when security is disabled. We look at this summary more closely:

```
select 0
```

[6] Taking the absolute value is necessary for comparative profiles, in which a summary's cum can be negative.

This sets [`SecurityManager.getClassContext`] as the current summary (call it s). BOTTLENECKS prints s, the base and cum of s, and s's top and bottom extensions (with a number assigned to each one). In this case, there are no bottom extensions, and the top extension

```
[ SecurityManager.checkMemberAccess ;
  SecurityManager.getClassContext ]
```

(call this s') has a much higher cum than the other choices. This is extension number 0, and we look at it more closely:

```
select 0
```

This sets s' as the current summary and BOTTLENECKS prints s' and its metrics and extensions; the length of s' is greater than 1, so BOTTLENECKS also prints the summaries that result from trimming the top or bottom method of s' (the former is s again).

The next step is to extend s' at the top. In general, we repeatedly extend the current summary by choosing the extension with the highest cum. This process continues until all potential extensions have a low cum, or until there are many possible extensions, no one of which contributes substantially to the overhead. Note that, if we were using zooming, this process would be mostly automatic.

In this case, we continue extending at the top until we reach the summary on the left side of Figure 1. At this point, there are many top extensions (for various Trade3 commands) and none of them contribute substantially to the overhead. This summary contributes 25% of the total overhead, which is significant, so we decide that it is a bottleneck and label it:

```
label ‘‘bottleneck’’
```

3 Experience

This section presents our experience with BOTTLENECKS. Section 3.1 discusses bottlenecks we found in the implementation of the security model in IBM's WebSphere Application Server (WAS) and novel optimizations that target those bottlenecks. In Section 3.2, we evaluate how helpful BOTTLENECKS is for finding bottlenecks in two other object-oriented applications: the SPECjAppServer2002 [28] application server benchmark [7] and a program under development at IBM that is related to the optimization of XML.

[7] SPEC and the benchmark name SPECjAppServer2002 are registered trademarks of the Standard Performance Evaluation Corporation. In accord with the SPEC/OSG Fair Use Policy, this paper does not make competitive comparisons or report SPECjAppServer metrics. For more information, see http://www.spec.org.

3.1 Speeding up WAS security

WAS implements J2EE, which is a collection of Java interfaces and classes for business applications. Most importantly, J2EE implementations provide a *container* that hosts Enterprise JavaBeans (EJBs). In J2EE, applications are constructed from EJBs; the container is like an operating system for EJBs, providing services such as database access and messaging, as well as managing resources like threads and memory.

Among these services is security. Security is optional: containers like WAS can be configured to bypass security checks. Enabling security entails some overhead. For example, in our experiments, turning on full security reduced the throughput of IBM's Trade3 [29] benchmark by 30%.

To find the causes of this slowdown, we collected call-tree profiles of Trade3, running with security enabled and with security disabled, and compared the profiles with Bottlenecks, using the comparative implementation of the profile interface. The result was fourteen bottlenecks, which accounted for over 80% of the overhead of enabling security.

The rest of this section discusses four topics: what we mean by "WAS security" in this paper, our experimental setup, the bottlenecks we found in WAS security, and finally optimizations that target those bottlenecks.

Security in WAS For our purposes here and at a high level, WAS supports two kinds of security:

Java 2 security Java 2 security is the security model supported by Java 2, Standard Edition (J2SE) [15]. In this model, an application developer or system administrator associates permissions with Java code: that is, the model supports framing and answering questions of the form, "may this code access that resource?". For example, an administrator can use Java 2 security to prevent applets from accessing the local filesystem.

Global security Global security is the security model supported by Java 2, Enterprise Edition (J2EE) [14], with extensions specific to WAS. Along with user authentication, the model supports framing and answering questions of the form, "may this user invoke that code?" A system administrator associates *roles* both with methods and with users: a user may invoke a method only if a role exists that is associated with both the user and with the method.

Experimental Setup To measure WAS security overhead, we needed an application that uses security. We used IBM's Trade3 benchmark, which is a client-server application developed specifically to measure the performance of many features of WAS. Trade3 models an on-line stock brokerage, providing services such as login and logout, stock quotes, stock trades, and account details.

The standard version of Trade3 does not use WAS security, so we used a "secured" version, which defines one role that protects all bean methods. We assigned that role to two users and used one of those users to access the application; the other user was the WAS administrator.

Table 1. Characteristics of Trade3 ArcFlow profiles. **Nodes** lists the number of tree and leaf nodes in each call-tree profile; **Depth** lists the maximum and average (arithmetic mean) depth of tree nodes; **Out-degree** lists the maximum and average out-degree of interior nodes.

Security	Nodes		Depth		Out-degree	
	Total	**Leaf**	**Max**	**Mean**	**Max**	**Mean**
Disabled	413637	189386	135	59.9	69	1.84
Enabled	480994	230248	135	64.6	109	1.92

Table 2. Trade3 bottlenecks, and the time and number of BOTTLENECKS commands needed to find them. Time includes the user's "think" time.

Bottlenecks			Cost to find	
Number	**Coverage**	**Mean length**	**Minutes**	**Commands**
14	82.7%	14	32.4	151

Trade3 is a small application, with only 192 classes and 2568 methods. By comparison, WAS contains 23270 classes and 246903 methods. Still, Trade3 revealed significant security bottlenecks in WAS.

We analyzed the performance of Trade3 when hosted by version 5.0.0 of IBM's WAS implementation. We ran the benchmark on a single desktop computer, which had 1.5GB of RAM and a single 2.4GHz Intel Pentium 4 processor. The operating system was Red Hat Linux 7.3 [25], and the database system used was version 7.2 of IBM's DB2 [9]. The secured Trade3 also requires an LDAP server: we used version 4.1 of IBM's Directory Server [18].

To collect call-tree profiles, we used IBM's ArcFlow profiler [3]. ArcFlow builds a call-tree on-line by intercepting method entries and exits. ArcFlow can collect various metrics: we chose to collect executed instructions (as measured by the Pentium 4 performance counters) because ArcFlow can accurately account for its own perturbation of this metric. Time would have been a better metric, but all profilers available to us either perturb this metric too much or collect only flat profiles.

Bottlenecks in WAS Security Using ArcFlow, we collected two call-tree profiles of Trade3: one from a run with WAS security enabled and one from a run with WAS security disabled. Both runs were otherwise as identical as we could make them (for example, both runs performed the same number of transactions). Then, one of the authors used BOTTLENECKS to find paths with high security overhead.

Table 1 lists some characteristics of the call-tree profiles. The profiles are bushy and deep: in both cases, roughly half of the nodes are leaf nodes and the average depth of a node is around 60. Thus, these profiles are not human-readable; an analyst would need a tool to make sense of them.

Table 2 shows how effective the author was at finding bottlenecks in these profiles with Bottlenecks. The author is, of course, an expert at using Bottlenecks. The author was also familiar with some of the bottlenecks in Trade3, because he had found them earlier without the aid of Bottlenecks.

Bottlenecks worked well on Trade3. Over 80% of the security overhead of Trade3 is covered by just fourteen bottlenecks. Also, these bottlenecks are useful: by optimizing six of them, we obtained a 23% improvement in throughput with security enabled.

In addition, the author found the bottlenecks quickly. By contrast, before Bottlenecks existed, it took the author over a week to find fewer and less specific bottlenecks by studying call-graph profiles and source code.

Security Optimizations This section describes four optimizations inspired by the bottlenecks that we found in the Trade3 profiles. Together, these optimizations speed up Trade3 by 23% when security is enabled. Three of these optimizations were applied to WAS and apply directly to other WAS applications. In addition, all four optimizations exploit two general properties of Java security, and we believe that similar optimizations could be applied in other applications, or perhaps automatically in a just-in-time compiler.

Both properties have to do with the high redundancy of security checks. We distinguish two kinds of redundancy. A security check exhibits high *temporal redundancy* if the check makes the same decision based on the same data several times in close succession (for example, a check that repeatedly tests if the same user may invoke bean methods that are protected by the same role). A security check exhibits high *spatial redundancy* if the check frequently makes the same decision because it is reached by the same code path (for example, a `checkPermission` that is executed repeatedly in the same calling context).

Optimizations that exploit temporal redundancy are based on caching. The results of an expensive check are stored in a cache, which is indexed by the data that form the basis of the decision. The cache is consulted before making the check: if the decision is in the cache, the check is avoided. Note that caching is effective only when cache hits are sufficiently frequent and the cost of cache lookups and maintenance is sufficiently cheap.

Optimizations that exploit spatial redundancy are based on specialization. The frequent code path is copied, and the expensive check is replaced with a cheaper version tailored specifically to that path. Specialization is effective only when the benefit of the tailored check outweighs the cost of duplicating code.

Table 3 summarizes our optimizations. The CheckRole and DBReuse optimizations exploit temporal redundancy, while the GetCredentials and Reflection optimizations exploit spatial redundancy. The ease of implementing these optimizations varied: Reflection took less than an hour, DBReuse and GetCredentials less than a day. The CheckRole optimization took a few days, because we wrote three versions before finding a fast cache implementation. In general, the temporal optimizations were harder to implement than the spatial optimizations, because the temporal optimizations required implementing a cache. For the spa-

Table 3. Optimizations suggested by Trade3 bottlenecks.

Optimization	Kind	Lines	Estimated Improvement	Comment
CheckRole	temporal	216	0.07	Remove redundant role checks on bean method access. In WAS.
DBReuse	temporal	343	0.04	Remove redundant comparisons and hashes for database connections. In WAS.
GetCredentials	spatial	114	0.06	Remove doPrivileged and checkPermission on a hot path. In WAS.
Reflection	spatial	157	0.11	Replace field access via reflection with direct access. In Trade3.

tial optimizations, most of the lines we changed were merely copied from one place to another.

For each optimization, we estimated the potential improvement in throughput by comparing the overhead of the bottleneck(s) that the optimization exploited to the total security overhead. The estimates in Table 3 assume that all of the bottleneck's overhead can be eliminated, and that the instruction counts reported by ArcFlow correlate well with execution time. In fact, optimizations cannot normally eliminate all overhead, and ArcFlow misses the overhead of I/O. Thus, these estimates are overestimates; see below for the actual improvements in throughput.

Detailed descriptions of our optimizations follow.

CheckRole The J2EE security model allows an administrator to associate roles with methods and with users. When global security is enabled, WAS inserts an access check before each bean method call. If there is a role that is associated with both the current user and with the method, then the call is allowed; otherwise, it is forbidden.

Our analysis found that role-checking is a bottleneck that accounts for 16% of the instruction-count overhead of security when running Trade3. By instrumenting the code, we discovered that these checks have high temporal redundancy, which we exploited by caching. Here is pseudocode for role-checking:

```
Routine CheckRole(User u, Method m) =
  Return UserRoles(u) ∩ MethodRoles(m) ≠ ∅
```

Our optimization introduces a *decision cache* (DC):

```
Routine CachingCheckRole(User u, Method m) =
  If DC.lookup(u, m) Then Return true
  ElseIf CheckRole(u, m) Then (DC.add(u, m) ; Return true)
  Else Return false
```

The decision cache is indexed by the user and by a set of roles. For fast lookups, we modified the WAS code to ensure a correspondence between users and objects representing users; there was already a correspondence between methods and objects representing methods. With this implementation, the cache can check equality simply by comparing references.

To further speed lookups, the decision cache is hierarchical. The cache contains a hash table that maps methods (which vary more frequently than do users) to 4-element arrays. Given a user and a method, a lookup starts by finding the 4-element array for the method; if the array exists, then the lookup scans it for a match with the user. The alternative of composing the user and method into a hash table key is significantly more expensive.

The CheckRole optimization is effective only when role checks have high temporal redundancy. Temporal redundancy is high for Trade3 because there is only one user and one role, so checks necessarily repeat. However, even if multiple users were configured, temporal redundancy in Trade3 should remain high, because each page request corresponds to seven bean method calls. We do not know if such behavior is common among J2EE applications: the more common it is, the more widely applicable is the CheckRole optimization.

DBReuse Because creating a database connection is expensive, WAS reuses them: when a transaction starts, WAS assigns it a connection from a pool of available connections; when a transaction completes, its connection is returned to the pool so that it can be used again.

In J2EE, a Subject represents information about a user or other entity. When security is enabled, WAS associates a Subject with each database connection. This association complicates connection pooling, because WAS must ensure that a database connection that is opened on behalf of one user is never used on behalf of a different user. Our analysis found that the security checks necessary to provide this guarantee comprise three bottlenecks, which account for about 10% of the instruction-count overhead of security when running Trade3.

Once again, we used caching to remove this overhead. Most of the overhead of the check was related to checking equality of Subjects and computing hash codes for Subjects. These operations are expensive because Subjects contain private credentials, which cannot be read without first passing a permission check. Our caches avoid this expense by remembering the results of equality checks and hash code computations.

GetCredentials Like DBReuse, this optimization speeds up an operation that depends on reading private credentials. However, while DBReuse is a temporal optimization, this optimization is spatial. The optimization is an instance of a general technique for removing Java 2 permission checks.

Permission checking in Java 2 security is complicated, but we can suppose that it provides two primitives: `checkPermission` and `doPrivileged`.

The `checkPermission` method receives a permission object as its only argument, and walks the call-stack to verify that the code has the permission represented by the object. Intuitively, on a call of `checkPermission`, the Java

runtime visits each call on the call-stack, visiting each callee before its caller. At each call, the runtime consults a table (prepared by the administrator) to decide whether the invoked method has the permission or not. If the runtime finds a method that does not have the permission, it raises an exception.

A `doPrivileged` call is used to cut off the stack walk. If, while walking the stack, the runtime finds a `doPrivileged` call, it stops the walk. Thus, the walk's outcome cannot depend on the calling context of the `doPrivileged`.

Our optimization exploits this property. The following pseudocode illustrates the optimization, as we applied it to the bottleneck on the right side of Figure 1, which accounts for 13% of the instruction-count overhead of security when running Trade3:

```
Class PrivilegedClass
  Routine Invoker = doPrivileged ...GetCredentials()...
Class CheckingClass
  Routine GetCredentials() =
    checkPermission(constant) ; Return the secret credentials
```

We can optimize this code as follows:

```
Class CallingClass
  private Object secret
  private bool succeeded := false
  Routine checkSecret(Object o) = Return secret = o
  Routine Invoker =
    If succeeded Then creds := GetCredentials(secret)
    Else doPrivileged
           creds := GetCredentials()
           succeeded := true
Class CheckingClass
  private bool succeeded := false
  Routine GetCredentials(Object o) =
    If ¬ CallingClass.checkSecret(o) Then
      checkPermission(constant)
    ElseIf ¬ succeeded Then
      checkPermission(some constant)
      succeeded := true
    Return the secret credentials
```

The optimized code performs a full security check just once; if the check succeeds (the common case), then any further calls perform a fast security check. The fast check uses a secret object, known only to `CallingClass`, to verify that the caller is `CallingClass.Invoker`. Because the secret is private to `CallingClass` and escapes only to `GetCredentials`, it cannot be forged. So, if attacking code calls `GetCredentials`, its permissions will be checked in the normal, safe way.

Table 4. Performance benefit of the optimizations.

Optimization	Throughput (pages/s) Insecure	Secure	Improvement Insecure	Secure	Security Overhead
Original	101.1 ± 0.3	71.4 ± 0.2	0.00	0.00	29%
CheckRole	98.7 ± 0.4	74.1 ± 0.2	-0.02	0.04	25%
DBReuse	101.7 ± 0.3	72.3 ± 0.2	0.01	0.01	29%
GetCredentials	100.1 ± 0.3	71.1 ± 0.2	-0.01	0.00	29%
Reflection	102.8 ± 0.2	74.1 ± 0.3	0.02	0.04	28%
All	103.4 ± 0.3	88.1 ± 0.1	0.02	0.23	15%

There is a caveat: once the permission checked by `GetCredentials` is granted to `CallingClass.Invoker`, it must never be revoked. In this instance, the caveat is not problematic—if the code in question did not have the permission, then WAS would not work.

There are other ways to implement this optimization safely. First, the runtime could detect such redundant security checks and rewrite the compiled code (as we rewrote the source code) to avoid them. This solution would require no source changes at all and would automatically optimize away other occurences of the pattern. Second, one could add a module system to Java (such as MJ [8]), which would allow a programmer to say statically that `GetCredentials` may only be called by `Invoker`. Finally, instead of passing the secret as a parameter of `GetCredentials`, the secret could be stored in a thread-local variable. This last approach avoids changing the signature of `GetCredentials` but is slow on many JVMs, for which accessing thread-local storage is expensive.

Reflection Java's reflection API allows programs to ask the runtime for the methods, fields, and other attributes of classes. Reflection is inherently expensive, and code that must be fast should avoid using it. Reflection is especially expensive when security is enabled, because the runtime must check that code that requests attributes of a class has appropriate permissions.

Our analysis found that Trade3 uses reflection unnecessarily on the code path on the left side of Figure 1, which accounts for 25% of the instruction-count overhead of security when running Trade3.

Benefits of Security Optimizations Table 4 shows the performance of the original WAS and Trade3 code and the performance benefit of each optimization in isolation, all safe optimizations together (that is, all optimizations except GetCredentials), and all optimizations together. For each optimization, we rebuilt the Trade3 system from scratch and measured performance with security enabled and with security disabled. To obtain a measurement, we warmed up the system by requesting 50000 pages, and then measured the time for Trade3 to satisfy 20000 page requests, repeating the latter measurement ten times. The **Throughput** columns of the table report the mean and probable error (that

is, 50% confidence interval) of these measurements, assuming a normal distribution. The **Improvement** columns report the mean improvement in throughput with respect to the unoptimized code. Finally, the **Overhead** column reports the overhead of enabling security after applying each optimization.

Overall, we obtained a 23% improvement in throughput (with security enabled) with all optimizations.

The performance benefit of all optimizations together exceeds the benefit of the optimizations separately. This is a reproducible effect, which we are not sure how to explain.

In general, the optimizations achieve a little more than half of the estimated improvement of Table 3. This indicates that the ArcFlow profiles, which measure executed instructions instead of time, miss some security overhead. Unfortunately, we are unaware of any profiling tools for Java that combine context-sensitivity (crucial for finding these bottlenecks) with sufficiently accurate measurements of execution time.

3.2 Other Applications of Bottlenecks

This section evaluates the effectiveness of Bottlenecks on two more applications:

SPECjAppServer2002 SPECjAppServer2002 is a client-server application developed specifically to measure and compare the performance of J2EE application servers. At runtime, SPECjAppServer2002 consists of an application server, the EJBs that are hosted by the application server, a database system, and a driver and supplier emulator. The SPEC reporting rules require that the emulator run on a different machine than the other components, but, in our tests, we ran all components on the single computer described above.

XML This is an internal IBM program, written in Java, and related to the optimization of XML. For this application, we had an ArcFlow profile from the developers but neither an executable nor the source code. Thus, this application represents an extreme case of performance analysis with very little information.

We used Bottlenecks to analyze ArcFlow profiles of both applications. We collected the SPECjAppServer2002 profile ourselves; the XML profile for XML was given to us by one of its developers.

Table 5 lists some characteristics of these profiles. The SPECjAppServer2002 profile was large, with over one million nodes, while the XML profile was relatively small. Both profiles are bushy and deep: in both cases, roughly half of the nodes are leaf nodes and the average depth of a node is 34.2 (SPECjAppServer2002) and 21.9 (XML). As was the case for the Trade3 profiles, these profiles are not human-readable.

Table 6 shows how effective one of the authors was at finding bottlenecks in these profiles with our tool. Once again, the author is an expert at using

Table 5. Characteristics of ArcFlow profiles. **Nodes** lists the number of tree and leaf nodes in each call-tree profile; **Depth** lists the maximum and average (arithmetic mean) depth of tree nodes; **Out-degree** lists the maximum and average out-degree of interior nodes.

Application	**Nodes**		**Depth**		**Out-degree**	
	Total	**Leaf**	**Max**	**Mean**	**Max**	**Mean**
SPECjAppServer2002	1096416	516173	77	34.2	74	1.89
XML	24321	11107	86	21.9	62	1.84

Table 6. Bottlenecks found for each application, and the time and number of Bottlenecks commands needed to find them. Time includes the user's "think" time.

Application	**Bottlenecks**			**Cost to find**	
	Number	**Coverage**	**Mean length**	**Minutes**	**Commands**
SPECjAppServer2002	13	35.8%	8.7	50	251
XML	13	88.7%	6.2	29.5	143

Bottlenecks, although he was not familiar a priori with the bottlenecks in SPECjAppServer2002 and XML.

Bottlenecks worked well on XML. The author quickly found thirteen bottlenecks that cover almost 90% of the executed instructions of XML. Also, these bottlenecks are useful: we reported them to one of the XML developers, who told us that they accurately identified the expensive paths in that application.

Bottlenecks was less effective on the SPECjAppServer2002 profile. The author quit analyzing the profile after about an hour: at this point, he had found thirteen bottlenecks that accounted for only 36% of the executed instructions. These bottlenecks were also less useful, primarily because the ArcFlow profile measures only executed instructions. By using a sampling-based profiler that measures time accurately but ignores calling context, the author found that SPECjAppServer2002 is I/O-bound, not compute-bound. Because ArcFlow does not measure execution time spent waiting on I/O, it is unlikely that optimizing the bottlenecks we found would significantly improve execution time.

4 Related work

Our profile interface can be implemented for any profile that associates metrics with execution paths. A number of tools produce profiles that satisfy this assumption. Program tracers like QPT [7] record the control flow of an entire execution. Ball-Larus path profilers [4] record intraprocedural, acyclic control-flow paths. Interprocedural path profiles [21] generalize Ball-Larus path profiles. Whole program path profilers [17] record an execution trace in a compact, analyzable form. Calling-context trees [2] are space-efficient cousins of the call-tree profiles we use in this paper. ArcFlow [3], which we used for the experiments in this paper, constructs call-tree profiles on-line by intercepting method entries

and exits. Stack sampling [12, 13] is an alternative, lower overhead method. Finally, Ball, Mataga and Sagiv show that intraprocedural paths can be deduced, with significant accuracy, from edge profiles [6].

Many other tools exist for analyzing profiles. The closest to BOTTLENECKS is Hall's call-path refinement profiling [12, 13]. The summary construction functions in Figure 3 are essentially special cases of Hall's call-path-refinement profiles, and Hall also describes a tool for navigating call sequences. However, our work differs from Hall in several ways. First, while our profile interface can be implemented for any profile that associates metrics with execution paths, Hall assumes a specific stack-sampling profiler. Second, Hall addresses the issue of overlap differently, by extending the user interface with the ability to prune away time spent either in or not-in given call paths. Finally, Hall's tools do not support comparing profiles.

Another closely related analysis tool is the Hot Path Browser [5] (HPB), a visualizer for Ball-Larus path profiles. HPB graphically shows overlap among intraprocedural Ball-Larus paths and allows the user to combine profiles by taking their union, intersection, and difference.

Fields and others [11] use *interaction cost* to find microarchitectural bottlenecks, while we use overlap to find bottlenecks in large applications. Overlap and interaction cost our closely related—in fact, they are arithmetic inverses of one another. In their work, interaction cost was important because processors perform tasks in parallel. In our work, overlap was important because the same execution-cost occurs in the context of many different call-sequences.

BOTTLENECKS assumes that profiles are not flat and can be analyzed off-line. For efficiency, performance analysis tools for large-scale parallel systems often violate one or both of these assumptions. For example, Paradyn [22] avoids collecting large amounts of data by interleaving measurement with interactive and automatic bottlenecks analysis. Paradyn's search strategy is top-down, although their DeepStart stack-sampling heuristic [26] can suggest starting points that are deep in the call-tree. Other tools for parallel systems, such as HPCView [20] and SvPablo [10], gather only flat profiles. Insofar as these compromises are necessary for analyzing parallel systems, they pose an obstacle to applying tools like BOTTLENECKS to such systems.

Our security optimizations that exploit temporal redundancy rely on identifying checks that repeatedly operate on the same data. When we suspected temporal redundancy, we verified it by instrumenting the code. Object equality profiling [24] might have discovered these opportunities more directly.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 74–89. ACM Press, 2003.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the*

ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, pages 85–96. ACM Press, 1997.
[3] Real-time ArcFlow. http://www.ibm.com/developerworks/oss/pi.
[4] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.
[5] Thomas Ball, James R. Larus, and Genevieve Rosay. Analyzing path profiles with the Hot Path Browser. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.
[6] Thomas Ball, Peter Mataga, and Shmuel Sagiv. Edge profiling versus path profiling: The showdown. In *Symposium on Principles of Programming Languages*, pages 134–148, 1998.
[7] Tom Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(3):1319–1360, July 1994.
[8] John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 241–254. ACM Press, 2003.
[9] IBM DB2 Universal Database. http://www.ibm.com/db2.
[10] Luiz DeRose and Daniel A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, September 1999.
[11] Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, pages 228–242, December 2003.
[12] Robert J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.
[13] Robert J. Hall. CPPROFJ: Aspect-capable call path profiling of multi-threaded Java applications. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 107–116, September 2002.
[14] Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee.
[15] Java 2 Platform, Standard Edition (J2SE). http://java.sun.com/j2se.
[16] Joseph M. Juran and A. Blanton Godfrey, editors. *Juran's Quality Handbook*. McGraw-Hill, New York, New York, USA, fifth edition, 1999.
[17] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 259–269. ACM Press, 1999.
[18] IBM Tivoli Directory Server. http://www.ibm.com/tivoli.
[19] Thomas J. McCabe and G. Gordon Schulmeyer. *Handbook of Software Quality Assurance*, chapter The Pareto Principle Applied to Software Quality Assurance, pages 178–210. Van Nostrand Reinhold Company, 1987.
[20] John Mellor-Crummey, Robert Fowler, Gabriel Marin, and Nathan Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of SuperComputing*, 23:81–101, 2002.
[21] David Melski and Thomas W. Reps. Interprocedural path profiling. In *Computational Complexity*, pages 47–62, 1999.
[22] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam,

and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[23] Objective Caml. http://www.ocaml.org.

[24] Robert O'Callahan and Darko Marinov. Object equality profiling. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 313–325, October 2003.

[25] Red Hat Linux. http://www.redhat.com.

[26] Philip C. Roth and Barton P. Miller. Deep start: A hybrid strategy for automated performance searches. In *Euro-Par 2002*, number 2400 in Lecture Notes in Computer Science, August 2002.

[27] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *Proceedings of TOOLS Europe*, 2001.

[28] SPECjAppServer2002. http://www.specbench.org/jAppServer2002.

[29] IBM Trade3 J2EE Benchmark Application. http://www.ibm.com.

[30] WebSphere Application Server. http://www.ibm.com/websphere.