RICE UNIVERSITY

Transactional Profiling for Multi-Tier Applications

by

Anupam Chanda

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Alan L. Cox
Associate Professor of Computer Science,
Rice University

John Mellor-Crummey
Associate Professor of Computer Science,
Rice University

Kartik Mohanram
Assistant Professor of Electrical and
Computer Engineering, Rice University

Willy Zwaenepoel
Professor and Dean, School of Computer
and Communication Sciences, EPFL

Houston, Texas
April, 2007
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3256671
Copyright 2007 by ProQuest Information and Learning Company. All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346
Transactional Profiling for Multi-Tier Applications

Anupam Chanda

Abstract

This dissertation is concerned with performance debugging of multi-tier applications, such as those commonly found in web services and dynamic-content web sites. A multi-tier application receives an external stimulus (request), executes the request and returns the response. A transaction begins with a request and ends with the corresponding response. It is executed by a series of different stages of the application. Existing tools and techniques for profiling such applications are not general enough to track and profile transactions in a generic multi-tier application. We propose transactional profiling that provides a general solution to this problem. We provide novel algorithms and techniques to track and profile transactions that flow through shared memory, events, stage queues or via inter-process communication using messages. We also measure interference among concurrent transactions. The thesis of this work is that transactional profiling addresses the limitations of existing profiling techniques for multi-tier applications.

Transactional profiling works by tracking transaction execution paths in an application. Essentially, it extends the concept of call path profiling to a distributed environment. Additionally, it introduces new methods to track transactions for shared memory communication, and communication through events and stage queues. Transactional profiling is mainly composed of the following: tracking transactions in a multi-tier application, and associating profile data with transactions. Further, trans-
actional profiling captures the interference among concurrent transactions, e.g., transactions waiting to acquire locks.

The contributions of this thesis are the following. We propose the transactional profiling model for profiling multi-tier applications. We describe the design and implementation of Whodunit, our prototype transactional profiler. Using Apache and MySQL we demonstrate the correctness of our proposed algorithm for tracking transaction flows through shared memory. We demonstrate the use of Whodunit in obtaining the transactional profile of web servers, a web proxy cache, a bookstore application, and a bulletin board application. Whodunit-inspired optimizations increased the peak throughput of the bookstore by almost 3x and the bulletin board by almost 2x. We measured Whodunit's overhead on the performance of these applications and found it to be small — less than 6% in all cases.
Acknowledgments

This dissertation would not have been possible without the help and support of Dr. Alan L. Cox, my advisor. I would like to thank him for his guidance and encouragement. This thesis has been the product of the countless discussions and brainstorming sessions that I had with Alan. Alan’s superlative knowledge and expertise in computer systems has fueled a constant hunger in me to learn and understand the details of the design and implementation of the various aspects of computer systems.

I would like to thank Dr. Willy Zwaenepoel for his guidance and advice. Willy raised several important questions that helped shape the direction of my research. The amazing clarity of his thought process and his remarkable acumen in identifying the fundamental principles of complex systems has immensely helped me to generalize the crux of my research work. I am grateful to him for traveling to Houston for my thesis defense.

Many thanks to Dr. John Mellor-Crummey and to Dr. Kartik Mohanram for their insights and suggestions. John’s profound knowledge in the area of profiling has helped me to understand the concepts behind profiling, and has educated me on the multitude of related work and prior art in this area. Kartik’s suggestions have been instrumental in broadening the scope of my dissertation.

I would like to thank Dr. John Mellor-Crummey, Nathan Froyd, and Dr. Robert J. Fowler for providing me with the implementation of the csprof profiler that I used in the implementation of Whodunit.

My sincere thanks to the faculty members and the research staff of the Computer
Science department for the excellent academic program. Many thanks to the past
and present members of the administrative staff of the Computer Science department
— Darnell Price, Iva Jean Jorgensen, Lena Sifuentes, Rhonda Guajardo, BJ Smith,
Melissa Cisneros, Donna Jares, Bel Martinez, and Maria Melendez — for their help.

It was a pleasure working with the fellow graduate students in the Computer
Science department. I would like to thank Khaled Elmeleegy, Santashil PalChaudhuri,
Amit Saha, Anirban Mandal, Rajnish Kumar, Sitaram Iyer, Sameh Elnikety, Animesh
Nandi, Atul Singh, Johnny Ngan, and Anwis Das for making graduate study at Rice
University a memorable phase of my life.

I would like to thank Vicky Xu, my manager at VMware, Inc. Vicky has been
very understanding and supportive of my work on completing the dissertation in
addition to my responsibilities at VMware. I would also like to thank my colleagues
at VMware, especially Gustav Wibling. Gustav has been a wonderful technical lead
on the project I have been working on and graciously shouldered my responsibilities
while I took a leave of three days from the office for my thesis defense.

I would like to thank my parents and brother for their support and belief in
me. They have constantly encouraged me through the ups and downs of graduate
study which now has finally reached its culmination. They share the same sense of
achievement that I feel with the completion of this dissertation.

Finally, I would like to thank my wife, Surupa, for her support and encouragement.
Her wonderful companionship has enriched my life and has helped make a better
person out of me. She has been there for me all the time — sharing my frustrations,
failures, success, and joy — and I cannot thank her enough for that. This dissertation
would not have been completed without her constant caring and support.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations x
List of Tables xii

1 Introduction 1
1.1 Motivation 2
1.2 Thesis Statement 4
1.3 Thesis Summary 5
1.4 Thesis Contributions 6
1.5 Organization 7

2 Transactional Profiling: Background and Overview 9
2.1 Profiling of Stand-Alone Applications 9
2.1.1 Call Graph Profiling 9
2.1.2 Call Path Profiling 11
2.2 Transactional Profiling Overview 12
2.2.1 Definitions 12
2.2.2 Examples 13
2.2.3 Transactional Profiling 15
2.2.4 Challenges in Transactional Profiling 15
3 Transactional Profiling Through Shared Memory

3.1 Assumptions ............................................. 19
3.2 Algorithm ............................................. 20
3.3 Detecting Transaction Flow ............................... 23
  3.3.1 An Example from Apache 2.x ............... 23
  3.3.2 Further Discussion ............................... 24
3.4 Avoiding False Positives ............................... 25
3.5 Transactional Profiling ................................. 29
3.6 Implementation ....................................... 29

4 Transactional Profiling Through Events and Stages 31

4.1 Transaction Flow Through Events ..................... 31
  4.1.1 Tracking Transaction Flows ..................... 33
  4.1.2 Debugging of Event-Driven Programs ........... 35
4.2 Transaction Flow Through Stages in SEDA ............ 35
4.3 Implementation ....................................... 37

5 Operating Systems Support for Tracking Transactions
and Transactional Profiling Across Distribution 38

5.1 Motivation ............................................. 39
5.2 Causeway Design ..................................... 41
  5.2.1 Metadata ......................................... 42
  5.2.2 Interfaces ....................................... 42
  5.2.3 Support for Propagation of Metadata .......... 44
6 Transaction Crosstalk and Profiling Heterogeneous Layers of Execution

7 Case Studies
7.3.1 Whodunit’s Overhead on Haboob ......................... 72
7.4 TPC-W ................................................. 72
  7.4.1 Whodunit’s Overhead for TPC-W ...................... 77
7.5 tBoard ................................................. 78

8 Related Work .............................................. 82
  8.1 Research Tools ......................................... 82
  8.2 Commercial Tools ...................................... 85

9 Conclusions ................................................. 87
  9.1 Future Research Directions .............................. 89

A Appendix on Causeway ..................................... 90
  A.1 Example Use of Causeway: Multi-tier Priority Propagation ............... 90
     A.1.1 Metadata Access .................................. 91
     A.1.2 Application ....................................... 91
     A.1.3 Experiment ....................................... 91
     A.1.4 Results .......................................... 92
     A.1.5 Discussion ....................................... 97

Bibliography .................................................. 98
## Illustrations

1.1 A 3-stage multi-tier application. .................................................. 2

3.1 Code snippet from Apache 2 web server showing producer-consumer pattern of shared memory access. ........................................... 23
3.2 Program with shared state that does not affect individual transactions. 26
3.3 Memory allocator in a multithreaded program. ............................ 28

4.1 Modifications to libevent to support transactional profiling. ........ 33
4.2 Modifications to a SEDA-based program to support transactional profiling. .......................................................... 36

5.1 Call path trees of the caller (left) and the callee. ....................... 54
5.2 Transaction flow connecting the caller and the callee. ................. 55

6.1 Translating a native profile sample to a source-level profile sample. . 62

7.1 Transactional profile of Apache under the Rice web workload. ....... 66
7.2 Transactional profile of Squid under the Rice web workload. ........ 69
7.3 Transactional profile of Haboob under the Rice web workload. ...... 71
7.4 Average response time for AdminConfirm, BestSellers & SearchResult transactions under the original and the optimized cases.  
                                                                                       76

7.5 Throughput (in transactions/minute) under browsing mix workload with and without caching.  
                                                                                   77

7.6 tBoard throughput (in transactions/minute) with and without caching  
                                                                                       81

A.1 Response Time Distribution (Sorted in Descending Order) for Search-Request Interaction (High Background Load)  
                                                                                     94

A.2 Response Time Distribution (Sorted in Descending Order) for Search Request (Moderate Background Load)  
                                                                                         96
Tables

5.1 The Causeway API ................................. 42
5.2 The Callback Interface .......................... 44
5.3 Causeway Overhead (getpid test) .............. 49
5.4 Causeway Overhead (pipe test) ................. 49
5.5 Transfer Points for System-visible Channels in the FreeBSD Kernel . 51
5.6 TPC-W Throughput (interactions/minute) for Shopping Mix ........... 53

7.1 Execution time of Apache’s critical sections (in machine cycles) for
    the different modes of execution. .................. 68
7.2 MySQL CPU profile (%) and mean crosstalk waiting times for the
different TPC-W transactions for browsing mix workload with 100
    concurrent clients. ................................. 73
7.3 Transaction crosstalk matrix (lock wait time in milliseconds) for
    AdminConfirm, BuyConfirm, and BestSellers transactions for
    browsing mix workload with 100 concurrent clients. .......... 74
7.4 Peak throughput (transactions/minute) of TPC-W under various
    profiling tools. ................................. 78
7.5 MySQL CPU profile (%) for the different tBoard transactions for a
    workload with 10 concurrent clients ............ 80
A.1 Average Response Time and 95% Confidence Interval (in milliseconds) for the TPC-W Interactions under High Background Load 93
A.2 Average Response Time and 95% Confidence Interval (in milliseconds) for the TPC-W Interactions under Moderate Background Load 95
Chapter 1

Introduction

This dissertation is concerned with performance debugging of multi-tier applications, such as those commonly found in server applications, dynamic-content web sites and web services. For instance, a dynamic web server commonly consists of a web server, an application server and a database. Figure 1.1 shows an example of a dynamic content web site composed of three stages — a web server, an application server and a database server. In such applications, a client request is passed between different stages. Stages may be distributed over multiple machines, reside on the same machine, and sometimes belong to the same process (e.g., multiple threads executing a single request). In our example of a dynamic web server, the web server acts as an HTTP front-end, which invokes the business logic in the application server. The business logic may issue multiple queries to the database, before it returns to the web server. The web server then constructs the HTTP response to the client. We call the execution of a client request through a series of multiple stages a transaction — not to be confused with ACID (Atomicity, Consistency, Isolation, Durability) transactions.

Performance debugging of multi-tier applications continues to be an interesting and challenging problem [AMW+03, BDIM04, IBB+04, RWM+06]. Profile data on the individual stages of an application usually fail to provide insight into the behavior of the entire application as a whole. For instance, in the above example of a three-stage dynamic web server, profiling the database back end in isolation does not provide the profile data for the different transactions executed by the application. To solve
this problem, we aim to provide an end-to-end profile of transactions in a multi-tier application.

1.1 Motivation

A number of tools have been developed for performance debugging of single-component applications. Call path profiling [Hal92, HG93] is a technique that has been particularly useful. A call path profiler records the resource usage (CPU usage, cache misses, etc.) associated with each call path in a program. Such a call path profiler is superior to tools like gprof [GKM82], which approximately attributes a procedure’s resource usage to its callers based on the calling frequency from its callers. Attributing resource usage to full call paths is much more accurate.

Such single-component tools can be used on the individual components of multi-tier applications, and may yield useful information, but they often do not tell the whole story. For instance, in the case of a 3-stage dynamic web site consisting of a web server, an application server, and a database server, a single-component profiler may tell us that the database CPU is 100% utilized, and that much of the CPU time is spent inside the sorting routine, but it does not tell us which type of dynamic content request caused this behavior in the database. Running multiple single-component profilers, one for each stage, cannot shed any further light on this problem, because each profiler sees its stage in isolation, and does not have information about other
stages. The developer is then often left with no other solution than to manually try to reconstruct the relationships between the execution in different stages. An end-to-end profile of the entire application would address the above shortcomings of single-component profilers, thereby immensely aiding the performance debugging process of multi-tier applications.

Often the components of a multi-tier application utilize communication channels other than messages over sockets. For example, in a multi-threaded program, threads communicate via shared memory. Event-handlers in an event-driven program communicate via events or continuations. And, stage threads in a Staged Event-Driven Architecture (SEDA) [WCB01] program communicate through stage queues. An end-to-end profile model for a generic multi-tier application should be able to construct the execution flow of requests for such communication paths among the components in a multi-tier application.

The industry has been quite active in the area of distributed profiling. A few profiling tools are commercially available that can profile distributed Java 2 Platform Enterprise Edition (J2EE) [Sunc] applications, e.g., PerformaSure by Quest [QSa], Optibench from Performant [Per], and Borland ServerTrace [Cor]. Economic viability and implementation feasibility has limited the availability of such tools to the J2EE platform only. As a result, such tools are not directly applicable to applications composed of many widely used programs like Apache and MySQL [lam]. Further, these tools have limited functionality because they cannot follow transactions through events, SEDA stages, or shared memory.

The research community has also been very active on the topic of performance debugging of distributed applications. Project 5 has investigated performance debugging of a distributed system of black boxes over a local area network [AMW+03] and over a wide area network [RWM+06]. Project 5 analyzes performance at the "box"
level — it does not find the bottleneck causing points in the application program within a box. Magpie [BDIM04] measures per-request resource usage in a distributed application. Magpie requires knowledge of application-specific event schema, that can only be provided by an expert, and relies on operating system support that generates detailed traces of program execution events. Not all operating systems provide the necessary tracing support that Magpie requires. Further, requiring detailed knowledge of application behavior is a limitation in itself.

Often concurrent transactions in a multi-tier application interfere among themselves while acquiring locks. While a transaction holds a lock in exclusive mode, it causes the other transactions that need to acquire that lock to wait while it completes the execution of its critical section. Prior art in distributed profiling has not modeled this interference among concurrent transactions.

I propose transactional profiling, a novel profiling model that profiles the different transactions in a multi-tier application and provides an end-to-end profile of the application as a whole. While doing so, transactional profiling addresses the above mentioned limitations of the state-of-the-art in profiling multi-tier applications. In particular, transactional profiling tracks communication among stages through shared memory, events, SEDA stage queues or message passing over sockets or pipes. On each stage of the application, transactional profiling associates profile data with the type of request executing in the stage. Further, transactional profiling captures the interference among concurrent transactions due to lock contention.

1.2 Thesis Statement

My thesis is that transactional profiling can be achieved in a transparent and efficient manner. Transactional profiling does not require any modifications (under most cases)
to a multi-tier application — support for transactional profiling can be implemented in the operating system and system libraries. Additionally, transactional profiling can be realized at a small cost to the application's performance. To support this thesis, I have designed and implemented a transactional profiling tool. Using this tool, I obtain end-to-end profiles of several multi-tier applications. I show how the profile data led to the performance optimization of two dynamic content web sites, viz., a bookstore and a bulletin board. I also measure the performance overhead of this tool on these applications.

1.3 Thesis Summary

The goal of transactional profiling is to automatically extend performance debugging tools from single-component to multi-component applications. For instance, in the above example of a three-stage dynamic content web site, transactional profiling not only tells us where time is spent in the database server, but it also tells us what type of dynamic content request to the web server caused this behavior in the database server. Similarly, transactional profiling tells us not only that threads in the database server spend a lot of time waiting for locks, but it also tells us which web requests caused the database threads to hold or wait for these locks.

Transactional profiling works by tracking the causal contexts of execution across different stages of the applications. A stage can be a process, a thread, an event handler (in an event-driven application) or a stage worker thread (in an application using the staged event-driven architecture [WCB01]). Stages communicate among themselves via shared memory, events, stage queues or messages over sockets or pipes. The constituent features of transactional profiling are:

- It tracks causal contexts of transactions across stages for communication via
shared memory, events, stage queues or messages. It then associates the resource usage of the application with each causal context, thereby providing per transaction resource usage.

- For concurrent transactions, it measures the interference among different transactions due to lock contention.

1.4 Thesis Contributions

Transactional profiling in a multi-tier application is complicated by a number of factors. First, execution of a transaction is often distributed across process and machine boundaries. Second, an executing stage may be a process, a thread, an event-handler or a stage thread, e.g., SEDA [WCB01]. A transaction needs to be tracked across all such stages. Third, threads may pass a particular request between themselves via shared memory. Communication through shared memory is generally harder to track than inter-process communication (IPC) via message passing. Fourth, some of the stages may be written in a mix of native and interpreted code — for instance, often a web/application server is written in C and PHP [lam]. For most users, information is most useful if presented at the PHP source level, and not in terms of the code implementing the PHP interpreter engine. Finally, due to concurrent execution, a transaction may cause another transaction to wait, e.g., due to lock contention. Our solution relies on maintaining this knowledge about the request on behalf of which code executes at any point in a multi-tier application through the entire life of the request. My thesis addresses all these complications to realize transactional profiling.

The following are my thesis contributions:

- Transactional profiling is a novel profiling model that tracks causal contexts
of executing transactions across different stages, viz., threads, processes, event-handlers and stage threads. I provide a set of techniques to track causal contexts across communication among different stages.

- I provide novel algorithms to track the flow of transactions from one stage to another via shared memory. Tracking transaction flow through shared memory is a hard problem because, generally, it is not visible to the operating system. My solution is based on observing the memory access pattern to the shared memory region.

- Transactional profiling associates resource usage to transactions in a multi-tier application. It extends a single-component profiling tool to a distributed environment. I have extended call path profiling to multi-tier applications in my implementation. The design and implementation of my transactional profiling tool serves as a guide to extend any single-component profiling tool to a distributed environment.

- Finally, a novel aspect of transactional profiling is measuring the interference among concurrent transactions. Previous work had measured interference among threads only. My work is an improvement in the sense that it can measure interference among transactions that execute in a distributed environment.

1.5 Organization

The rest of this dissertation is organized as follows. In Chapter 2, I present a brief overview and background of transactional profiling. I describe the design and implementation of transactional profiling for communication among stages via shared memory and through events and stage queues in Chapters 3 and 4 respectively. Chapter 5
describes operating system support for tracking transactions across distribution and the design and implementation of transactional profiling for communication among stages via message passing over sockets and pipes. I present the design and implementation of transaction crosstalk (to present and measure the interference among concurrent transactions) and how transactional profiling handles heterogeneous layers of execution in Chapter 6. Chapter 7 describes case studies of using transactional profiling on several multi-tier applications. I present a survey of related work on profiling distributed systems in Chapter 8. Finally, in Chapter 9, I draw the conclusions of this dissertation.
Chapter 2

Transactional Profiling: Background and Overview

In this chapter I first provide background information on the fundamentals of profiling of stand-alone applications. Then I describe a brief overview of transactional profiling.

2.1 Profiling of Stand-Alone Applications

There is a rich body of work in the realm of profiling stand-alone applications. By far, call graph profiling [GKM82] and call path profiling [Hal92, HG93] are the two most well known profiling techniques for stand-alone applications. I describe these two profiling models in this section. As mentioned earlier, transactional profiling extends the notion of call path profiling in a multi-tier application. However, transactional profiling's underlying idea of establishing causal connections among the profiles in a multi-tier application can be applied to extend any of the stand-alone profiling techniques to achieve end-to-end profiling in a multi-tier application.

2.1.1 Call Graph Profiling

`gprof` is a well known and widely available profiling tool that implements call graph profiling [GKM82]. `gprof` collects three kinds of profile information about an application. First, it counts the number of times a routine, a statement or a basic program block is executed. Second, it measures the execution time in different routines of the program. Third, it gathers information about the caller-callee relationships among the routines during program execution.


**gprof** introduces instrumentation in the program to count the number of executions of a routine, a statement or a program block. Such instrumentation is introduced by compiling the program with special flags (profiling flags, "-pg," on most platforms). The compiler inserts a call to a monitoring routine in each routine's prologue. This monitoring routine counts the number of times a routine is executed. The monitoring routine also records the arc (from the caller to the callee) that activated a program routine. The count of number of times a routine is executed is associated with this arc. The total number of times a routine is called is obtained by summing the counts associated with all arcs leading to it. The monitoring routine, thus, constructs the dynamic call graph of the program at run-time.

**gprof** employs a sampling technique to measure the execution time of routines. Under a sampling technique, an “alarm clock” interrupts the program at regular intervals. The interrupts are fielded by a monitoring routine that records the program counter when the interrupt occurred. The monitoring routine maintains a histogram of program counter values and the associated sample counts in memory at run-time.

When the program terminates, the table of (caller-callee) arcs and the histogram of program counter samples are written to a file. A post-processing tool inspects the dumped profile data, and generates a flat profile — a histogram of program routines and associated samples, and a call graph profile — a call graph of routines in the program and associated sample counts.

A limitation of call graph profiling is that it assumes that the cost of a called routine (callee) is independent from the call site, and thus, call graph profiling apportions the cost of the callee to the calling routines in the ratio of the number of times they called the callee. This assumption of call graph profiling is often erroneous — call path profiling addresses this limitation.
2.1.2 Call Path Profiling

Hall first proposed the technique of call path profiling [Hal92] to address the above shortcoming of the existing profiling techniques including call graph profiling. In particular, call path profiling measures and associates resource usage with call paths in a program. A call path is the complete dynamic context leading to a particular point in the program, e.g., the ordered list of call sites leading to a specific point in the program. The prior art in profiling techniques, though produced useful profile information, left a lot of "detective" work to the user to identify performance bottleneck causing areas of a program. Call path profiling removes the need for detective work by directly identifying the dynamic contexts in the program that consume more resources than other parts of the program.

Intel's VTune [Int] is a call path profiler; it employs instrumentation of the program to enable profiling. This instrumentation can lead to significant dilation of the program's execution time. VTune documentation recommends sparing use of call path profiling rather than using it on the whole program [Int]. Apple's Shark [App] is a statistical call path profiler: it is based on sampling the program call stack. However, it does not count the number of times a procedure is called from the different calling contexts. Hence, it cannot estimate the cost of a procedure call based on its calling context.

The above limitations are addressed in csprof [FMCF05], a recently implemented call path profiler tool. csprof has low, controllable overhead. It is a statistical call path profiler — periodically sampling the program call stack. csprof can count call path edges, thereby having the ability to estimate the average cost of a procedure called from any calling context. csprof can profile unmodified program binaries that have been compiled with the highest levels of optimization. Since it does not instru-
ment the program, its overhead on the running time of the profiled program is within a few percent. Transactional profiling extends call path profiling to a distributed application. Since call path profiling is at the core of transactional profiling, any call path profiling tool may be used in an implementation of transactional profiling. In our implementation of a prototype transactional profiler, we chose csprof as its call path profiler core.

2.2 Transactional Profiling Overview

2.2.1 Definitions

A stage in a multi-tier application executes the whole or a part of a transaction. In this thesis, a stage may be a process, a thread, an event-handler or a stage worker thread (SEDA). A stage can be an entire program or a program can be composed of multiple stages. Stages communicate among themselves via channels which can be sockets, pipes, shared memory, events or SEDA stage queues. A multi-tier application receives an external stimulus, which we term a request, performs an execution to process this request and returns a response. A transaction begins with a request and ends with the corresponding response. The transaction context is the execution path of a request through the different stages. At any point during a transaction’s execution, its transaction context captures the complete execution history through the different stages of the application.

For a stand-alone program, we define the execution path as the call path of the program. In a multi-tier application a transaction is executed by multiple stages, and each of the stages has its own execution path for the transaction. Transaction context is the complete execution path for all the stages concatenated in the order of their execution. We illustrate a transaction context below with examples of stages
communicating through sockets or pipes, shared memory, events and SEDA stage queues.

2.2.2 Examples

Typically web services are comprised of three stages — a web server, an application server and a database server. During the execution of a transaction these stages send and receive messages over sockets. Execution of a transaction begins at the web server when a request arrives. Its transaction context in the web server is simply the call path in the web server program. The web server sends a message to the application server; after that the application server continues the execution of the transaction. The transaction context at the application server is the call path of the web server program at the point where it sends the message, concatenated with the call path in the application server program. Similarly, the transaction context at the database server includes the call paths through the web server, the application server and the database server, in that order. When the database server sends back a message to the application server, we identify that the response for a previous message from the application server has arrived and that execution of the transaction has resumed at the application server. Likewise, we identify the response message from the application server to the web server.

A good example of communication among stages through shared memory occurs in the Apache 2.x web server. A listener thread in Apache receives a request and stores it in a shared data structure. A worker thread obtains this request from the shared data structure and continues execution of the transaction. The transaction context of the request in the listener thread is the thread's call path in the program. The transaction context of the request in the worker thread is its call path prefixed by the call path of the listener thread, at the time when it stored the request in the
task queue.

In an event-driven application, execution of a request begins with an initial event-handler, e.g., an event-handler in a web server that accepts incoming connections. The transaction context of a request while being executed by the initial event-handler is simply the call path in the program. An event-handler may set up a continuation to continue execution when an event occurs in the future. When such an event-handler executes, its transaction context is the call path of that event-handler, prefixed by the transaction context of the event-handler that had set up the continuation for this current event. To illustrate, in a web server a request gets executed by an event-handler that accepts incoming connections (accept-handler), one that reads the HTTP request (read-handler), and one that sends back the HTTP response (write-handler). When the accept-handler executes, the request’s transaction context is simply the call path. When the read-handler executes, the transaction context is the concatenation of the accept-handler’s call path followed by the call path of read-handler. Similarly, the transaction context when the write-handler executes includes the call paths of the accept-handler and the read-handler, in that order, followed by the current call path in the write-handler.

Execution in a SEDA-based application proceeds through multiple stage worker threads that communicate via stage queues. A stage worker thread executes a request and then produces an event in the stage queue connecting it to the next thread. Execution of the request then continues with the next stage. The execution begins with the initial stage of the program, during which the transaction context of a request is the same as the call path. At any stage of the system, the transaction context is the call path of the stage prefixed by the transaction context of the previous stage at the point where it produced the event in the stage queue between these two stages that serves as input to the current stage. Thus, transaction context at any stage includes
the call paths through all previous stages.

2.2.3 Transactional Profiling

We now state how transactional profiling is performed. In call path profiling [Hal92, HG93], each sample in a program's profile data gets annotated with its call path. Similarly, in transactional profiling, we annotate a profile sample with its transaction context. That is, when a profile sample occurs, we find the transaction context at that point of execution and annotate the profile sample with that transaction context. For example, in a three-stage web service application, a profile sample on the web server gets annotated by its call path in the web server, a profile sample on the application server gets annotated by its transaction context, which consists of the call paths through the web server and the application server, and a profile sample on the database server gets annotated by the call paths through the web server, the application server and the database server. These annotations allow us to measure the resource usage of each request type on all the stages of the system [CCZ07]. As an example, we may identify the web server request types that cause high CPU utilization at the database server in a three-stage application.

2.2.4 Challenges in Transactional Profiling

The challenge in transactional profiling is to follow transactions through the different stages. From the above discussion we can infer that the stages exhibit a producer-consumer relationship when passing transactions between themselves. In some cases this producer-consumer relationship is explicit. For example, in the case of message passing, the send operation corresponds to the produce, and the receive operation to the consume. In event-driven programs creating a continuation corresponds to the produce, and executing a new event-handler to the consume. SEDA stages are similar:
a produce inserts an event in the stage queue, and a consume removes it. In such cases where the producer-consumer relationship is explicit, tracking the transaction is relatively easy. It suffices to remember the transaction context at the (explicit) produce point, and to pick it up at the (explicit) consume point. Additionally, if the produce and the consume operations are encoded as library functions, for instance, as in message passing libraries, event libraries, or SEDA stage classes, then the code for tracking transactions can be hidden inside those libraries. If not, for example with a hand-coded event-driven server, then the application needs to be modified, but the modifications are modest. The next three chapters describe our solution for tracking transactions across stages.

Even in shared memory, if there are explicit functions by which tasks are produced and consumed, the same approach can be followed. The situation is more challenging when transactions are passed implicitly through shared memory, in other words when there is a sequence of instructions that operate on shared memory but without a high-level indication that the instructions implement transactions being passed from one stage to another. This coding style is often used in practice, for instance in the Apache 2.x web server, and therefore we need to handle such scenarios. In this case the producer-consumer relationship needs to be inferred from the shared memory accesses. We explain our solution to this problem in the next chapter.

2.2.5 Implementation Overview

We have implemented Whodunit, a prototype transactional profiler, that addresses the above challenges and presents an end-to-end profile of a multi-tier application. We now provide a brief overview of Whodunit’s implementation.
Whodunit

Whodunit's run-time system is implemented as a library that is pre-loaded to initiate the profiling of a program. An initialization routine creates the profiler's state and initiates profiling. The program is then loaded, and Whodunit profiles its execution. When the program exits, Whodunit finalizes its state and writes the profile data to disk. In a final presentation phase, Whodunit stitches together the profiles from the application stages using transaction context information.

At run-time Whodunit annotates profile data with the transaction contexts of executing transactions. Whenever a profile sample occurs, Whodunit records the profile data and annotates it with the transaction context of the currently executing transaction. Whodunit computes transaction context by tracking transactions through shared memory, events, SEDA stages and IPC.

At the core of Whodunit is a call path profiler. The call path profiler core is responsible for collecting the profile data that Whodunit annotates. We used cssprof [FMCF05], a recent implementation of a call path profiler, as the core of Whodunit. cssprof is a statistical profiler. It periodically samples the program and collects profile data. cssprof maintains its call path profile in an efficient data structure, the Calling Context Tree (CCT) [ABL97].

Whodunit annotates the root of a CCT with transaction context information. Thus, each CCT is labeled to be used for a different transaction context. When a stage receives a request from another stage, Whodunit computes the transaction context of the sending stage's call path. At the receiving stage Whodunit then finds the CCT with that transaction context as its annotation, and instructs the call path profiler core to accumulate profile samples in that CCT. If Whodunit does not find a CCT with this transaction context, it creates a new, initially empty CCT, and labels
it with this transaction context. Thus, Whodunit maintains the correct mapping between the currently executing transaction and the CCT wherein profile samples are collected.
Chapter 3

Transaction Profiling Through Shared Memory

Threads in a multithreaded program access shared data for a variety of purposes. One purpose may be to pass transactions as described in the previous chapter. Another may be access to shared application data in a critical section. Yet another form of access may occur in the implementation of a shared memory allocator. In transactional profiling our goal is to develop an algorithm that detects shared memory access for the purpose of passing transactions between threads, and avoids including other forms of shared memory access.

In this chapter, first, we state our assumptions under which we are able to detect transaction flow through shared memory. Then, we describe our algorithm, explain its operation, and show how it finds transaction flow and avoids false positives. Finally, we describe a mechanism to obtain transactional profiling under transaction flow through shared memory.

3.1 Assumptions

Accesses to shared data structures always occur in critical sections protected by locks. For every instance of transaction flow there is one thread that produces the data (producer) and another thread that consumes that data (consumer). Finally, the producer thread updates the shared data structure with data that it computes prior to entering the critical section — it does not generate the data inside the critical section. Conversely, the consumer thread obtains data from the shared data structure
and uses it after exiting from the critical section.

Threads in a multi-tier application have preset roles. They are either producers or consumers of a resource, but not both producers and consumers of the same resource, e.g., listener and worker threads in a server. In a SEDA [WCB01] program a thread consumes events from its input queue and produces events to its output queue to be processed by the subsequent stage's thread. Thus, a thread is a consumer of its input queue resource and a producer of its output queue resource, but not a producer and consumer of the same resource.

Finally, we assume that each different resource is protected by a separate lock. For example one lock protects a free memory pool resource shared among threads, another lock protects a work queue resource between listener and worker threads, and so on.

### 3.2 Algorithm

During execution application data resides in memory or in registers. The virtual address space of a process is the name space of all memory locations a program accesses. We can uniquely name a register $\text{reg}$ of thread $t_i$ by annotating it as $\text{reg}_i$. The union of the virtual address space and the name space of annotated registers is the complete name space of all locations where application data reside. We associate a location $\text{addr}$ in this name space (where $\text{addr}$ is either a memory location or a thread-register) with a transaction context which we maintain in a dictionary $(\text{addr}, \text{ctxt})$ indexed by $\text{addr}$. At any point in time a location $\text{addr}$'s dictionary entry either has no transaction context, a valid transaction context, or $\text{invctxt}$ which is a special value signifying an invalid context. Initially the dictionary is empty, i.e., no $\text{addr}$ has an associated transaction context.
Our algorithm analyzes the instructions in critical sections and performs the following actions for each MOV memory operation. A MOV memory operation moves data from a register or a memory location $addr_1$ to another register or a memory location $addr_2$. Assume thread $t_i$ executes a MOV operation in a critical section. We lookup the dictionary structure to find the transaction context associated with $addr_1$. If dictionary entry $(addr_1, ctxt)$ exists, we update the dictionary with an entry $(addr_2, ctxt)$, i.e., $addr_2$ gets associated with the transaction context $ctxt$. If $addr_1$ has no associated context, we compute the transaction context $ctxt_{t_i}$ of thread $t_i$ and associate it with $addr_2$, i.e., update $addr_2$'s dictionary entry as $(addr_2, ctxt_{t_i})$.

For any instruction in a critical section that is not a MOV instruction but modifies the value contained in location $addr$, we associate $invctxt$ with $addr$. Examples of such instructions include assigning an immediate value to a location, an arithmetic operation, incrementing (or decrementing) the value at a location, and so on.

A producer updates the shared data structure with some value that it computes before entering the critical section. The source location $addr_1$ of such value is some memory location or the thread's live registers on entry to the critical section. Since a location gets associated with a transaction context only inside a critical section, the source location $addr_1$ has no associated transaction context when the critical section is entered. When the value in $addr_1$ is moved to another location $addr_2$, we associate $addr_2$ with the executing thread's transaction context. When this occurs, we infer that the executing thread has produced a value in $addr_2$.

A consumer moves a value from a shared location $addr_1$ to a local location $addr_2$ and then uses the value from $addr_2$ after exiting the critical section. The location $addr_1$ is associated with the producer's transaction context when the producer moves the value it produces to $addr_1$ (as explained above). That context gets associated with $addr_2$, when the value in $addr_1$ is moved to $addr_2$. The consumer uses the value
from location $addr_2$ after exiting the critical section. When this event occurs, we infer that the consumer has consumed the value from $addr_2$ (because $addr_2$ has an associated transaction context).

Producers and consumers may also move elements previously produced in the shared data structure to new locations. For example, in a priority queue implementation both producers and consumers move elements in the queue to maintain the priority queue properties. Our algorithm automatically detects that. Assume an element $elem_1$ is stored at an address $addr_1$ in the shared queue. Also assume that $ctxt_1$ is the transaction context associated with $addr_1$. When a memory operation (or a sequence of memory operations) in a critical section moves $elem_1$ from location $addr_1$ to $addr_2$, $addr_2$ is associated with $addr_1$'s transaction context, i.e., $ctxt_1$. Subsequently, when $elem_1$ is consumed from the location $addr_2$, we find that the transaction context associated with $addr_2$ is $ctxt_1$.

A location $addr$ may be used for different purposes at different times. In particular, a location $addr$ may be used for transaction flow at one point in time but not another. Consequently, when we update the transaction context for $addr$ we remember the lock object corresponding to the executing critical section. We flush the transaction context associated with $addr$ if we find that it is being accessed from a critical section protected by a different lock object than the lock protecting the critical section that last updated $addr$'s transaction context.

Finally, we keep a list of producers and a list of consumers for every lock object. When we detect that a thread is producing a resource we add it to the list of producers for the lock object protecting that resource. Similarly when we identify that a thread is consuming a resource we add it to the list of consumers for the associated lock object. The first time we find that these two lists have a common member, we infer that transaction flow is not occurring for shared memory accesses protected by this
lock object.

3.3 Detecting Transaction Flow

3.3.1 An Example from Apache 2.x

```c
ap_queue_push(fd_queue_t *queue,
               apr_socket_t *sd, apr_pool_t *p) { /* producer thread */
    pthread_mutex_lock(&queue->one_big_mutex);
    elem = &queue->data[queue->nelts];
    elem->sd = sd;
    elem->p = p;
    queue->nelts++;
    pthread_mutex_unlock(&queue->one_big_mutex);
}

ap_queue_pop(fd_queue_t *queue
             apr_socket_t **sd, apr_pool_t **p) { /* consumer thread */
    pthread_mutex_lock(&queue->one_big_mutex);
    elem = &queue->data[--queue->nelts];
    *sd = elem->sd;
    *p = elem->p;
    pthread_mutex_unlock(&queue->one_big_mutex);
    /* caller uses values in sd & p after return */
}
```

Figure 3.1: Code snippet from Apache 2 web server showing producer-consumer pattern of shared memory access.

First, we describe the execution of our algorithm on the shared memory access pattern of the code snippet from Apache 2.0, shown in Figure 3.1. The producer (ap_queue_push) stores sd & p at some location addr in the shared queue. When the
MOV operations that move these values from their prior locations on the stack to addr execute, our algorithm detects that no transaction context is associated with these stack locations (since no data was moved to these stack locations inside a critical section). So it associates the producer thread’s transaction context $ctxt_{prod}$ with the location addr. The consumer (ap_queue_pop) thread later reads the values sd & p from location addr in the shared queue and moves them to local variables on its stack. When that happens these local stack variables’ locations get associated with the transaction context $ctxt_{prod}$. When the consumer uses these values from the local variables after exiting the critical section, our algorithm detects that these locations have associated transaction contexts, and from the value of these transaction contexts ($ctxt_{prod}$) infers that a transaction flows from ap_queue_push to ap_queue_pop.

3.3.2 Further Discussion

FreeBSD’s sys/queue.h provides implementations of singly-linked and doubly-linked lists and queues [www]. We have verified the correctness of our algorithm on test programs involving producers and consumers using the different data structures implemented by sys/queue.h.

Sometimes producer and consumer threads use consistency-checking code for sanity-checking of the shared data structure. For example, after retrieving the item to be consumed, the consumer may set the source location in the queue to a constant value, say NULL, signifying that the item has been consumed. Similarly, the producer might check if the value of a queue location is NULL before inserting a produced item at that location. We observe that in such a case the value NULL is being transferred from the consumer to the producer, but we should not infer transaction flow from the consumer to the producer.

The memory operation that stores the value NULL in the shared queue structure
does not move a value from one location to another location. It moves an immediate value (NULL) to a location. As mentioned earlier, when that happens the location gets associated with invalid, the invalid context. Thus, the value NULL does not cause any transaction flow from the consumer to the producer. Further, the producer does not use the value NULL it retrieved from the shared queue after exiting from the critical section. Even if it did so, we find that an invalid context is associated with the location containing that value, and thus no transaction flow will be detected.

The same reasoning applies when a producer enqueues an item elem in an empty queue, a consumer dequeues that item, and then another consumer tries to retrieve an item from the queue but finds that the head of the queue is NULL. This can happen in a linked-list implementation of a queue. The producer would have initialized elem->next to NULL. The first consumer would have moved the value NULL from elem->next to the head of the queue. Thus, the location containing elem->next has an invalid context, which is transferred to the location containing the head pointer of the queue (by the first consumer). The second consumer retrieves the value NULL from the head of the queue which is associated with the invalid context and thus no transaction flow is (correctly) inferred.

Programs often use nested locks. Our algorithm analyzes all instructions that are in the critical section protected by the outermost lock. Thus, all internal critical sections are also analyzed.

### 3.4 Avoiding False Positives

Some shared memory accesses have similarities to the producer-consumer pattern but do not constitute transaction flow. The following two examples are common shared memory access patterns that do not constitute transaction flow.
static int count = 0;

pthread_mutex_t mtx;

main() {
    pthread_create(thd1, func1, ...);
    pthread_create(thd2, func2, ...);
}

func1() { /* thread 1 */
    for (;;) {
        /* execute transaction */
        pthread_mutex_lock(&mtx);
        count++;
        pthread_mutex_unlock(&mtx);
    }
}

c2() { /* thread 2 */
    for (;;) {
        /* execute transaction */
        pthread_mutex_lock(&mtx);
        count++;
        pthread_mutex_unlock(&mtx);
    }
}

Figure 3.2: Program with shared state that does not affect individual transactions.
In Figure 3.2 two threads share a variable that is used to count occurrences of some event. From the program structure it is clear that this shared variable does not convey transaction flow from one thread to another. However, the same memory location is accessed by the critical sections of different threads to update the shared state \texttt{count}. This aspect is similar to the producer-consumer pattern where threads access the same location in the shared queue to propagate data. This memory access pattern is representative of a class of shared memory access patterns that inspect or update shared state that is independent of request data. Another example of this type of shared memory access pattern is inspection or modification of database tables by different threads of a database server.

Both threads increment the contents of the location \texttt{addr\_count} that contains the value of the shared variable \texttt{count}. For any instruction, other than a \texttt{MOV} instruction, that modifies the contents of a location, the algorithm associates the location with the \texttt{invlctxt} transaction context. Since the location \texttt{addr\_count} is the only shared location between the two threads and it is associated with an invalid transaction context, we correctly infer that no flow of transaction occurs between these two threads.

The \textit{memory allocator} pattern also bears resemblances to the producer-consumer pattern. Many applications implement their own memory allocator as a performance optimization to fulfill the dynamic memory requirements of the application threads. Threads executing transactions dynamically allocate and free memory via calls to the memory allocator. It is typically implemented as a shared data structure protected by a lock or mutex — Figure 3.3 shows pseudo code of a simple memory allocator.

We observe that the memory allocator pattern (Figure 3.3) of shared memory access is isomorphic to the producer-consumer pattern of memory access — the \texttt{mem\_free} routine is analogous to the producer and the \texttt{mem\_allocate} routine is analogous to the consumer. Since our algorithm keeps a list of producer threads and
static void *mem_free_list;
pthread_mutex_t mtx;
do_work() { /* thread i */
  for (;;) {
    void *mem = mem_alloc();
    /* execute transaction */
    mem_free(mem);
  }
}

void *mem_alloc() { /* allocate memory */
  pthread_mutex_lock(& mtx);
  /* mem = get & remove head from mem_free_list */
  pthread_mutex_unlock(& mtx);
  return (mem);
}

void mem_free(void *mem) { /* free memory */
  pthread_mutex_lock(& mtx);
  /* append mem to mem_free_list */
  pthread_mutex_unlock(& mtx);
}

Figure 3.3: Memory allocator in a multithreaded program.
consumer threads for each resource, it detects when these two lists have a common member. The first time that happens in the above code, our algorithm infers that memory accesses for this resource do not constitute transaction flows.

### 3.5 Transactional Profiling

We propagate transaction contexts between threads where transaction flow occurs in shared memory by the following mechanism. Suppose we identify that thread $t_p$ produced an item that is consumed by thread $t_c$. Let $tc_p$ be the transaction context of thread $t_p$ when it produced that item at location $addr$. This transaction context is associated with $addr$. When $t_c$ consumes the item from location $addr$ we assign $tc_p$ to thread $t_c$. From this point onwards the concatenation of $tc_p$ followed by the call path of $t_c$ defines $t_c$'s transaction context. Subsequent profile samples of thread $t_c$ get annotated by its transaction context.

### 3.6 Implementation

The transaction flow detection algorithm analyzes the instructions in the critical sections of programs that are protected by locks or mutexes. Use of system-visible primitives like `pthread_mutex_lock` and `pthread_mutex_unlock` makes the critical sections visible to Whodunit. This is achieved by providing wrappers for the lock acquire and release operations in Whodunit. Once the critical sections are identified, we “trap” the instructions that are performed from the critical sections. We achieve this trapping by emulating the code within the critical section.

We use the QEMU processor emulator [Bel] for emulation of the critical section code. QEMU is actually more powerful than what we need — it is a full system emulator capable of emulating the processor, memory and I/O devices. We extract
from QEMU the CPU emulator core that emulates machine instructions. For this extraction we turn off QEMU’s emulation of the processor’s memory management unit (MMU), as we emulate the critical section code from within the address space of the emulated process and thus we do not need to perform MMU address translation in software — the hardware does that for us. To avoid any stack corruption, the emulator always runs on a stack separate from the stack of the emulated program.

After extracting the necessary pieces from QEMU to support emulation of critical section code, Whodunit executes the algorithm described earlier in this chapter to identify transaction flows through shared memory. As a performance optimization, Whodunit stops emulating critical sections corresponding to resources that do not cause transaction flow, and executes them natively. For example, Whodunit stops emulating critical sections related to resources with memory access patterns similar to a memory allocator.

To identify when (and what) a thread consumes, Whodunit must continue emulating a thread’s execution for a few instructions past the exit from a critical section. If all produce and consume operations are implemented as procedures, then a consumer uses the value in the return register(s) immediately after the call to the consume operation completes. We emulate for a maximum of MAX instructions after exiting the critical section. We assume that the consumer uses the value it wants to consume within this window of MAX instructions. In our current implementation we use 128 as the value for MAX.

When Whodunit identifies that an item is being produced by a thread, it associates the location where the item is being produced with the transaction context of the executing thread in a dictionary structure. When Whodunit infers that an item is being consumed by a thread, it assigns the transaction context associated with the location the item is being consumed from to the executing thread.
Chapter 4

Transactional Profiling Through Events and Stages

4.1 Transaction Flow Through Events

A transaction in an event-driven application is executed by a sequence of event handlers. Execution proceeds from one event handler to the next via events. At any point in a transaction's execution, the complete sequence of event handlers executed on its behalf constitutes its transaction context.

To illustrate, suppose at a particular point in the execution of an HTTP transaction, a web server has executed the event handlers accept_handler, read_handler and write_handler, in that order. accept_handler accepts incoming client connections, read_handler reads the request data, and finally write_handler writes the response back to the client. Then, the transaction context at that point in the execution contains the ordered list \([\text{accept}\_\text{handler}, \text{read}\_\text{handler}, \text{write}\_\text{handler}]\).

For different transactions, different sequences of event handlers are executed, establishing separate transaction contexts. For example, consider an event-driven DNS server. Two different transactions are possible in this application: one corresponding to a cache hit and the other corresponding to a cache miss. Typically, cache hit and cache miss events are handled by different event handlers. So, two different transaction contexts will be established for this application.

We provide a mechanism to track transaction flow through an event-driven application. A transaction flows from one event handler to the next via an event (also known as a continuation). An event handler produces a continuation, that is con-
sumed by the event handler that acts on this continuation. We associate a transaction context with an event. This transaction context may be conceptually thought of as a string representing the sequence of event handlers executing a transaction. Assume, at any point in the execution an event $e_i$ with transaction context $\alpha$ is being executed by its event handler $evh_i$. If a new event $e_j$ is created to continue execution of the transaction, then the current transaction context is the concatenation of $e_i$’s transaction context and its event handler $evh_i$, i.e., the ordered list $[\alpha, evh_i]$. This transaction context suffixed by the program’s call path constitutes the transaction when $e_j$ is executed. When the initial event handler is scheduled, its transaction context is simply the call path of the program.

An event handler may be scheduled and executed (consecutively) more than once for a transaction if the operation performed by the event handler does not complete in a single iteration. For example, an event handler that performs a read or write operation may need more than one iteration to complete. In this case, the transaction context contains successive occurrences of that event handler. For example, the sequence may look like $[evh_A, evh_B, evh_B, evh_B, ..., evh_C]$, where $evh_A, evh_B, evh_C$ are distinct event handlers. We collapse multiple consecutive occurrences of the same event handler in the representation of a transaction context. Since we think of event handlers as stages, it makes sense to group the consecutive executions of an event-handler together.

As event-handlers are executed for a transaction and the sequence of event-handlers grows, sometimes a loop (of length greater than 1) in the sequence is created. For example, when multiple requests are served on a persistent connection, the list may grow as $[evh_{accept}, evh_{read}, evh_{write}, evh_{read}, evh_{write}, ...]$ and so on, where $evh_{accept}, evh_{read}$ and $evh_{write}$ are event handlers that perform the accept, read and write operations of a transaction, respectively. However, a loop (of length greater
than 1) in the sequence of event-handles is similar to a request being issued by a stage and the response coming back to it (e.g., RPC between stages). In this case, we prune the suffix of the sequence of event-handles that closes the loop, e.g., $[evh_{accept}, evh_{read}, evh_{write}, evh_{read}]$ is pruned to $[evh_{accept}, evh_{read}]$.

This pruning mechanism is similar to removing multiple occurrences of procedure nodes from a call graph or call path due to recursion (or mutual recursion). This is not strictly necessary for profiling, and the complete transaction context may be useful for some applications, e.g., for debugging. However, for profiling, the pruning mechanism helps to generate a concise presentation of the profile data.

4.1.1 Tracking Transaction Flows

1. event_loop()
2. {
3.     for (;;) {
4.         event *ev = get_next_ready_ev();
5.         curr_tran_ctxt = concat(ev->ev_tran_ctxt,
6.                                 ev->ev_handler);
7.         ev->ev_handler();
8.     }
9. }
10. event_add(new_ev)
11. {
12.     new_ev->ev_tran_ctxt = curr_tran_ctxt;
13.     add_to_monitor_list(new_ev);
14. }

Figure 4.1: Modifications to libevent to support transactional profiling.
To track transaction flow we augment the event or continuation structure with a transaction context field, that is filled in when the continuation is produced. Next, the program needs to remember the current transaction context, so that it may initialize the transaction context of a new continuation, should the currently executing event handler produce one. These modifications can be hidden in an event library, such as **libevent** [Pro03]. An event-driven program that uses such an event library requires no modification at all for transactional profiling.

Figure 4.1 shows the modifications to **libevent** to perform transactional profiling. The event structure has been augmented with a field **ev_tran_ctxt** that captures the transaction context when the event is produced by an event handler. Lines 5 and 6 are added to the event loop. The library maintains a global list — **curr_tran_ctxt** (the current transaction context). Before an event handler is invoked, the event loop computes the current transaction context by concatenating the selected event’s **ev_tran_ctxt** and its **ev_handler**. Loops, if any, are eliminated at this point (line 5). When a new event is created and registered, its **ev_tran_ctxt** is initialized with the global **curr_tran_ctxt** (line 12).

If an event-driven program does not use a standard event library, then the above modifications must be applied to its event loop and the continuation creation mechanism.

Performance analysis data for each transaction context is maintained separately as follows. Assuming sampling-based profiling, at any profile sample, the transaction context of the event-driven program is obtained from the global list **curr_tran_ctxt**. The profile sample is then annotated with the value of this list. Profile samples belonging to different transaction contexts, thus, have different annotations and are managed and presented separately.
4.1.2 Debugging of Event-Driven Programs

The **eel** tool [CK05] facilitates control flow analysis and debugging of event-driven programs. This tool generates a call graph connecting event-handlers by performing a static analysis. Callbacks may not be registered via function pointers and declared to be persistent while using the **eel** tool, instead they have to be registered every time their event-handler completes execution. It employs static analysis to construct a call graph connecting event-handlers, and as such callback registrations are not allowed to be persistent or use function pointers. In contrast, transactional profiling executes at run-time, and hence, does not impose these restrictions. Finally, the **eel** tool distinguishes between each client-server connection (for debugging purposes), while transactional profiling groups all transactions of the same type and their profile data together.

4.2 Transaction Flow Through Stages in SEDA

The Staged Event Driven Architecture (SEDA) [WCB01] has been proposed as an infrastructure to build scalable Internet services. Consecutive stages in SEDA have a producer-consumer relationship; they communicate via a stage queue connecting them. Transactions can be tracked in SEDA by using the shared memory transaction flow algorithm of Chapter 3. However, it is easier to track transactions by instrumenting the SEDA middleware. We describe that mechanism here.

We associate a transaction context with each queue element that is passed between successive stages. At any stage, when a thread dequeues and starts executing a request element from its input queue, it computes its current transaction context by concatenating the queue element's transaction context and the currently executing stage. Loops in the transaction context, as with event-driven programs, may form at
this step, and are eliminated in the same manner as in event-driven programs. When a thread creates a new queue element and puts it in its output queue, the new queue element's transaction context is initialized with the executing thread's transaction context.

1. stage_loop()
2. {
3.   for (;;) {
4.     queue_elem *elem = dequeue_next(input_queue);
5.     curr_tran_ctxt = concat(elem->tran_ctxt,
6.                               CURRENT_STAGE);
7.   /* execute elem */
8.   }
9. }
10. enqueue_elem(new_elem, output_queue)
11. {
12.     new_elem->tran_ctxt = curr_tran_ctxt;
13.     enqueue(output_queue, new_elem);
14. }

Figure 4.2: Modifications to a SEDA-based program to support transactional profiling.

Figure 4.2 shows the modifications to a SEDA-based program to enable transactional profiling (lines 5, 6, and 12). Conceptually, transactional profiling in an event-driven program and a SEDA-based program are very similar. This similarity is exhibited in the pseudo-codes of Figure 4.1 and Figure 4.2. As with event-driven programs, the modifications to a SEDA-based program are limited to the library implementing the SEDA architecture, and as such, the application using a SEDA library does not need any modifications at all. Transactional profile is collected in a manner
similar to the way it is done for event-driven programs: the profile data is annotated by the executing thread’s transaction context.

4.3 Implementation

We have implemented the above mechanisms in the libevent [Pro03] library and the SEDA library [WCB01]. We augment libevent’s struct event with a field to store the transaction context. event_loop and event_add are modified to track the current transaction context and to propagate transaction context between events. SEDA defines its own linked list class to implement the input and output queues of a stage. We modify this class to associate transaction context with individual queue elements and to propagate transaction context between stages. SEDA’s stage event loop is modified to keep track of the current transaction context.
Chapter 5

Operating Systems Support for Tracking Transactions and Transactional Profiling Across Distribution

An integral component of transactional profiling is propagation of transaction context information across the components of a distributed multi-tier application. In our implementation of transactional profiling, Causeway provides this service. We describe Causeway, its design and implementation, and how we use Causeway in transactional profiling.

Causeway provides runtime support for the development of distributed *meta-applications* [CECZ05a, CECZ05b]. These meta-applications control or analyze the behavior of multi-tier distributed applications such as multi-tier web sites or web services. Examples of meta-applications include transactional profiling, multi-tier debugging, fault diagnosis, resource tracking, prioritization, and security enforcement.

Efficient online implementation of these meta-applications requires meta-data to be passed between the different program components. Examples of metadata corresponding to the above meta-applications are transaction context information, request identifiers, priorities or security principal identifiers. Causeway provides the infrastructure for injecting, destroying, reading, and writing such metadata.

The key functionality in Causeway is forwarding the metadata associated with a request at so-called transfer points, where the execution of that request gets passed from one component to another. This is done automatically for system-visible chan-
nels, such as pipes or sockets. An API is provided to implement the forwarding of metadata at system-opaque channels such as shared memory. In this chapter we describe the design and implementation of Causeway, and we evaluate its usability and performance.

5.1 Motivation

Often, systems to control or analyze the execution of multi-tier applications are written to perform tasks like multi-tier debugging, fault diagnosis, resource tracking, prioritization, and security enforcement. Examples include transactional profiling, Pinpoint [CKF+02], Magpie [BDIM04, IBB+04], and Domain and Type Enforcement (DTE) [BSS+95] for Unix systems. We term these and similar systems that control or analyze the execution of multi-tier applications as meta-applications.

Traditionally, there have been two approaches to writing such meta-applications: a log-based approach, and a metadata-passing approach. The log-based approach operates in two phases — first, execution events of the application are recorded in logs, and next, the log records are analyzed. Magpie [BDIM04, IBB+04] and TraceBack [ASM+05] are examples of systems employing this approach. The log-based approach cannot affect the execution of requests in an online manner because processing of a log record lags the corresponding execution event by a positive time delay. Additionally, the execution events on the different tiers belonging to the same request need to be identified and connected while processing the log records.

The metadata-passing approach propagates metadata — arbitrary, out-of-band data — in addition to request data along execution paths. The meta-application accesses and utilizes this metadata to achieve its goal. Often, the metadata also serves in connecting a request's execution events spread across the tiers of the system, e.g.,
if it contains a request identifier. Several examples of meta-applications using this approach exist in the literature, e.g., Pinpoint [CKF+02] and DTE [BSS+95]. Pinpoint and DTE use request identifiers and security principal identifiers as metadata respectively. These meta-applications use hand-crafted code to handle and propagate metadata.

Unlike the log-based approach, the metadata-passing approach can affect the execution of requests in an online manner, e.g., Real-Time CORBA [Jon] which propagates priorities among application components to affect scheduling. Hence, we adopt the metadata-passing approach to building meta-applications. The motivation behind Causeway is to provide a framework that makes development of meta-applications using this approach easier.

Causeway provides an interface to associate metadata with threads of control and facilitates the propagation of metadata across communication channels. Causeway aids the development of meta-applications by performing all necessary management to handle and propagate metadata. This obviates the need for hand-crafted code for the common requirements of different meta-applications employing the metadata-passing approach.

The alternative to Causeway, propagating metadata at application level, involves augmenting all application-level inter-process communication protocols — a tedious solution. By making propagation of metadata a system-level function, it becomes independent of the application-level communication protocol used. Further, in a multi-tier application, it is possible that some individual components are unaware of the presence of metadata or choose to ignore it. Consider a three-tier system, where the middle tier component is unaware of metadata. The front and the back-end tiers may still, however, need to access metadata. In this scenario, operating system support for metadata propagation is required in the middle tier.
Causeway performs automatic propagation of metadata across system-visible communication channels. Such channels are those implemented in the operating system kernel and system libraries, e.g., pipes and sockets. Augmented kernel and system libraries provide Causeway’s support for system-visible channels. Causeway provides an API to be called from application code to perform metadata propagation across system-opaque channels, e.g., shared memory. Support for system-opaque channels is the essential difference between Causeway and Stateful Distributed Interposition (SDI) [RS04].

5.2 Causeway Design

At an abstract level, Causeway works as follows. A request to an application is executed by one or more threads of control, possibly in one or more tiers. Threads exchange request data along communication channels, e.g., sockets, pipes and shared memory. Causeway’s interface supports injection, inspection, modification and removal of metadata. Metadata is assigned to a thread when it performs injection. When a thread sends request data to another thread along a channel, Causeway transfers metadata from the former thread to the latter. Support for metadata propagation is required at transfer points where an application thread sends to or receives data from a channel. In this way, metadata, once injected, is propagated along the request execution paths.

Causeway has two parts: (1) a set of interfaces that are used by applications to manage and utilize metadata, and (2) mechanisms that implement propagation of metadata. First, we describe the structure and composition of metadata.
5.2.1 Metadata

Metadata in Causeway consists of a two-tuple containing the metadata type and the metadata value. Examples of metadata types include transaction context, request priority, request identifier, and security principal identifier. Meta-applications can define new metadata types, if required.

5.2.2 Interfaces

Meta-applications can interact with Causeway in two ways — through an interface to inject and access metadata and through a callback interface in which Causeway calls handlers registered by the meta-application.

Metadata Interface

Causeway provides interfaces for injection, inspection, modification, and removal of metadata. These interfaces may be called from user-level or kernel-level.

Causeway manages metadata in a dictionary keyed by the address of the associated entity. An entity is either a thread of control or data that is read from or written to a channel. A thread’s metadata is propagated to the data written on a write operation, subsequently this metadata is propagated from the data to a thread performing a read operation. Further, a thread can remove metadata associated with itself or a data
entity. Table 5.1 shows the function signatures of the Causeway API. The Causeway API performs metadata operations in the following manner:

- **cw_type_query** retrieves the collection of all metadata types associated with `addr` in the `types` array of size `ntypes`. On successful completion, `cw_type_query` returns the number of metadata types retrieved and `-1` on error. The `types` array must be large enough to hold all the metadata types associated with `addr` otherwise an error is flagged.

- **cw_data_lookup** retrieves the metadata of type `mtype` associated with `addr`. It returns `0` on successful completion and `-1` on error.

- **cw_data_insert** inserts the given metadata `md` of type `mtype` and associates it with `addr`, overwriting any prior metadata of that type. It returns `0` on successful completion and `-1` on error.

- **cw_data_remove** removes any existing metadata of type `mtype` associated with `addr`. It returns `0` on successful completion and `-1` on error.

**Callback Interface**

Using Causeway’s callback interface the meta-application can register a `transfer-point` callback method. A transfer point is a point where data is read from or written to a channel by a thread. At a transfer point Causeway determines if the type of the metadata being passed has a callback method registered. If a callback method exists, it is invoked with the metadata as an argument. The callback method reads and possibly modifies the metadata. The callback method can call arbitrary operating system code, e.g., to change the priorities of threads.
typedef void (*callback_t)(struct cw_metadata **md, int mtype);

callback_t callback_method;

void reg_callback_method(int mtype, callback_t callback_method);

Table 5.2: The Callback Interface

The signatures of a callback method and the callback interface are shown in Table 5.2. A callback method is of type `callback_t`. The callback interface, `reg_callback_method`, registers a given callback method for a given metadata type at a transfer point.

5.2.3 Support for Propagation of Metadata

When a thread performs a write on a channel, the thread's metadata is associated with the data written into the channel. On a subsequent read on the channel by a thread, metadata is propagated from the data and assigned to the thread.

There are two ways metadata can be assigned to a thread — injection and propagation across a channel. Newly assigned metadata replaces the thread's existing metadata of the same type.

Transfer Points

Places where a thread writes to or reads from a channel are transfer points. Channels are of two types: system-visible channels that occur in the operating system kernel and system libraries, e.g., sockets and pipes, and system-opaque channels that occur in the application, e.g., shared memory. Causeway exports a Systems Programming Interface (SPI) consisting of a single function `cw_metadata_xfer` for the purpose of implementing transfer points. `cw_metadata_xfer` takes a source entity and a desti-
nation entity as arguments. It obtains the source entity’s metadata and assigns the obtained metadata to the destination entity. At a transfer point for either a system-visible or system-opaque channel, a single call to `cw.metadata.xfer` is performed.

5.2.4 System-visible Channels

For system-visible channels, the metadata transfer SPI is automatically called from an augmented kernel and system libraries to implement Causeway’s support for metadata propagation. Sockets and pipes are system-visible channels supported by Causeway. Further, for a multi-threaded program, metadata needs to be propagated between the user-level thread and the kernel-level thread on entry to and exit from the kernel because multiple user-level threads may be multiplexed on top of a kernel-level thread. Metadata propagation between a user-level thread and a kernel-level thread constitutes additional system-visible channels in Causeway. We enumerate below the transfer points for system-visible channels:

1. *User-level thread to kernel-level thread*: On entry to the kernel, Causeway transfers metadata from the user-level thread to the kernel-level thread running it.

2. *Kernel-level thread to user-level thread*: On exit from the kernel, Causeway transfers the kernel-level thread’s metadata to the user-level thread.

3. *Kernel-level thread to message*: When a kernel-level thread writes a message on a socket or a pipe, its metadata is transferred to the message.

4. *Message to kernel-level thread*: When a kernel-level thread receives a message from a socket or a pipe, metadata is transferred from the received message to the kernel-level thread.

These transfer points occur in the operating system kernel and the threading library.
Causeway handles sockets and pipes similarly. When a thread writes to a socket (or a pipe), Causeway associates metadata from the thread to the data written via the metadata transfer SPI described above. Similarly, on a subsequent read from the socket by another (or the same) thread, metadata is propagated from the data to the thread.

The above applies for LOCAL sockets only. For INTERNET sockets, data is encapsulated in IP packets for send and receive across sockets. Causeway encapsulates metadata, in addition to data, in the IP packets. For IPv4, Causeway encapsulates metadata in the IP header as IP options. In particular, Causeway defines a new IP option type, populates the IP header with the option type, length, and payload. At the receiver side, metadata, if any, is extracted from the received IP options. Since IP options can be a maximum of 40 bytes only, with 1 byte each for the type and length fields, via this mechanism Causeway can transfer at most 38 bytes of metadata in IP packets. This limit on metadata size is deemed enough for most practical purposes. This limitation is an artifact of Causeway's implementation and not its design. A general purpose tunneling protocol could be used to overcome this limitation, if required.

5.2.5 Shared Memory — System-opaque Channel

For system-opaque channels, the application must be modified to call the metadata transfer SPI to perform propagation of metadata. Causeway supports metadata propagation across shared memory — a system-opaque channel implemented in user-space. A transfer point needs to be inserted in the application where a user-level thread reads from or writes to shared memory. Producer-consumer is a popular model of shared memory usage. At an abstract level, the model works as follows. Producers and consumers share a buffer or queue of objects. A producer creates an object, acquires
a lock to enter the critical section, adds the object to the shared buffer or queue, and releases the lock. A consumer acquires a lock to enter the critical section, retrieves and removes an object from the shared buffer or queue, releases the lock, and then accesses the retrieved object. The use of system-supported synchronization primitives, like `pthread_mutex` or `pthread_rwlock`, simplifies the task of identifying the producer-consumer communication channels through shared memory.

Two transfer points, one in the producer code and the other in the consumer code are inserted. Both transfer points use the metadata transfer SPI. The producer transfer point associates the producer thread's metadata with the produced object. The consumer transfer point retrieves the metadata associated with the consumed object and propagates it to the consumer thread. Causeway provides a user-level library that exports the metadata transfer SPI and manages the metadata associated with shared memory objects.

5.2.6 Heterogeneity of Operating System Kernel and Hardware

It is quite common for a multi-tier application to be spread across machines running heterogeneous operating system kernels on diverse hardware platforms. The design of Causeway mandates that all inter-machine metadata propagation be typed and be transmitted in network byte order. This ensures correct interpretation of metadata at the receiver. Further, our implementation of Causeway in FreeBSD lays out a blueprint for its implementation in other operating system kernels. In Section 5.4.1 we list the transfer points in the FreeBSD kernel required for the system-visible channels. An equivalent set of transfer points is required in another operating system kernel, such as Linux.
5.2.7 Operating System Specific Meta-applications

Sometimes, parts of a meta-application may require modifications to the operating system kernel. Under such circumstances, the meta-application becomes operating system specific. For example, we implemented a distributed priority enforcement system on top of Causeway which may alter priorities of threads and processes in a system — an operating system specific task. Thus, this meta-application is operating system specific. On the other hand, if all we wanted in a meta-application is to tag identifiers with requests, it would require no operating system modification other than Causeway itself.

5.3 Microbenchmarks

In this section we quantify the overhead imposed by our implementation of Causeway at the transfer points for two system-visible channels. We chose light-weight applications to provide maximum exposure to Causeway's overhead. We wrote two microbenchmarks: the first measuring the overhead associated with the transfer points for metadata propagation between a user-thread and a kernel thread, and the second measuring the overhead for the transfer points for the pipe channel.

In the first microbenchmark, a process creates a pthread which invokes a getpid call. This test brings out the cost of metadata propagation across the user-kernel boundary, because on each entry to and exit from the kernel, metadata is transferred from user space to kernel and vice versa. We repeat the getpid call multiple times and measure its average cost. We perform this experiment under the following scenarios: (1) without inserting the transfer point, which is the base case, (2) inserting the transfer point but transferring 0 bytes of metadata, (3) transferring 1 byte of metadata, and (4) transferring 32 bytes of metadata.
<table>
<thead>
<tr>
<th>Description</th>
<th>Cost (machine cycles)</th>
<th>Cost (microseconds)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base case</td>
<td>7001</td>
<td>2.92</td>
<td>-</td>
</tr>
<tr>
<td>0 byte metadata</td>
<td>7841</td>
<td>3.27</td>
<td>12.0</td>
</tr>
<tr>
<td>1 byte metadata</td>
<td>9369</td>
<td>3.90</td>
<td>33.8</td>
</tr>
<tr>
<td>32 bytes metadata</td>
<td>9409</td>
<td>3.92</td>
<td>34.4</td>
</tr>
</tbody>
</table>

Table 5.3: Causeway Overhead (getpid test)

<table>
<thead>
<tr>
<th>Description</th>
<th>Cost (machine cycles)</th>
<th>Cost (microseconds)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base case</td>
<td>35782</td>
<td>14.9</td>
<td>-</td>
</tr>
<tr>
<td>0 byte metadata</td>
<td>36807</td>
<td>15.3</td>
<td>2.9</td>
</tr>
<tr>
<td>1 byte metadata</td>
<td>49858</td>
<td>20.8</td>
<td>39.3</td>
</tr>
<tr>
<td>32 bytes metadata</td>
<td>54383</td>
<td>22.66</td>
<td>52.0</td>
</tr>
</tbody>
</table>

Table 5.4: Causeway Overhead (pipe test)

Table 5.3 shows the results of the above experiment. The cost of `getpid` increased by about 840 machine cycles when a transfer point was introduced. We used a 2.4 GHz Pentium 4 Xeon, so this overhead translates to about 0.35 microseconds. This result shows the cost of having the Causeway framework but not using it to propagate any metadata. The overhead increased by about 1500 machine cycles or about 0.6 microseconds when transferring 1 byte of metadata. To transfer 32 bytes of metadata, the further increase in overhead was small: about 40 machine cycles or 0.02 microseconds. In relative terms, the overhead with respect to the base case ranged from about 12% to less than 35% to transfer metadata in the above test.

The results of the above experiment show that the overhead of using Causeway is small. The overhead of inserting a transfer point is less than half of a microsecond.
The overhead of transferring 32 bytes of metadata is about 1 microsecond, and the overhead scales well with increasing metadata size.

The second microbenchmark measures the cost of transferring 1 byte of data between two processes across a pipe. As before, we perform this experiment under the four scenarios used in the previous microbenchmark. Table 5.4 shows the result for the pipe test. The overhead of inserting a transfer point but passing no metadata is similar to that of the getpid test. The overhead of passing metadata is higher because the metadata is propagated across address spaces. Nevertheless, the overhead of propagating up to 32 bytes of metadata is less than 8 microseconds, a small amount. Finally, the overhead scales well with increasing metadata size. In this test, Causeway’s overhead ranged from less than 3% to about 52% over the base case.

Note that for the above measurements we could not use a microbenchmark consisting of a network server and client as the cost of sending messages over the network is several orders of magnitude higher than the overhead of Causeway in terms of absolute cost and we would not have been able to detect the overhead of Causeway with such a microbenchmark.

5.4 Evaluating Causeway

In this section we quantify the complexity involved in Causeway to insert transfer points for system-visible channels, and transfer points in an implementation of the TPC-W [Cou02] benchmark. We also measure Causeway’s overhead on TPC-W.

5.4.1 Transfer Points for System-visible Channels

Sockets, pipes, and user-level thread/kernel-level thread boundary are the system-visible channels supported by Causeway. Six transfer points in the FreeBSD 5.2 kernel
<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
<th>File name</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>User thread to kernel thread</td>
<td>kern/kern_kse.c</td>
<td>thread_user_enter</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel thread to user thread</td>
<td>kern/kern_kse.c</td>
<td>thread_userexit</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel thread to socket message</td>
<td>kern/uipc_socket.c</td>
<td>sosend</td>
</tr>
<tr>
<td>Kernel</td>
<td>Socket message to kernel thread</td>
<td>kern/uipc_socket.c</td>
<td>sorreceive</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel thread to pipe message</td>
<td>kern/sys.pipe.c</td>
<td>pipe_write</td>
</tr>
<tr>
<td>Kernel</td>
<td>Pipe message to kernel thread</td>
<td>kern/sys.pipe.c</td>
<td>pipe_read</td>
</tr>
</tbody>
</table>

Table 5.5: Transfer Points for System-visible Channels in the FreeBSD Kernel

support metadata propagation across these channels as shown in Table 5.5. The user thread to kernel thread and kernel thread to user thread transfer points are required if the application is multithreaded. The socket and pipe transfer points are required if the application performs interprocess communication. Transfer points within system- visible channels do not require reimplementation for each new application.

5.4.2 Transfer Points for Apache and MySQL

We used Causeway to propagate metadata in an implementation of the TPC-W [Cou02] benchmark. Our implementation of the TPC-W benchmark used the Apache web server (version 1.3.31) built with the PHP module (version 4.3.6) and the MySQL database server (version 4.0.16). The TPC-W interactions are implemented as PHP scripts.

Apache is a multi-process web server and does not use shared memory communication among the different processes. Thus, no transfer points are required in Apache.

MySQL is a multi-threaded program and it uses the libpthreads library on FreeBSD. Inspection of the MySQL source code revealed that though individual MySQL pthreads
access some shared data structure in a synchronized manner, there is no communication between threads to exchange data corresponding to a single request. In other words, a request in MySQL is executed in its entirety by a single thread. An incoming database connection is accepted by a listener thread and handed over to a worker thread. The worker thread reads the request, executes it and sends back the response. Hence, no transfer points are required in MySQL as well.

In TPC-W, Apache and MySQL exchange messages across sockets. MySQL uses user-level threads on top of kernel-level threads. Thus Causeway's support for metadata propagation across system-visible channels, viz., sockets, and user-level thread and kernel-level thread boundary, suffices for our implementation of TPC-W using Apache and MySQL. This support is provided in an augmented FreeBSD kernel.

We have implemented a multi-tier priority enforcing system using Causeway to perform global priority enforcement on the requests to TPC-W. We describe this use of Causeway in Appendix A.

### 5.4.3 Overhead of Causeway on TPC-W

We conducted an experiment to evaluate the overhead imposed by Causeway on our implementation of TPC-W under a realistic workload. We subjected TPC-W to a workload consisting of emulated clients exercising the shopping mix [Cou02] workload. Apache, MySQL and the load generator ran on separate machines. All the machines were 2.4 GHz Pentium Xeon with 2 Gigabytes of memory, and were connected by switched Gigabit ethernet. We varied the number of concurrent emulated clients and measured the throughput (interactions per minute) obtained from TPC-W. We compare the throughput obtained with the Causeway framework with that obtained without the Causeway framework (base case). Under Causeway we transferred 4 bytes of metadata across each transfer point for TPC-W. Table 5.6 shows the results.
<table>
<thead>
<tr>
<th>No. of concurrent emulated clients</th>
<th>Throughput (base case)</th>
<th>Throughput using Causeway</th>
<th>Causeway Overhead(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>89.4</td>
<td>89</td>
<td>0.45</td>
</tr>
<tr>
<td>50</td>
<td>424.8</td>
<td>411</td>
<td>3.25</td>
</tr>
<tr>
<td>100</td>
<td>844.2</td>
<td>826.4</td>
<td>2.11</td>
</tr>
</tbody>
</table>

Table 5.6: TPC-W Throughput (interactions/minute) for Shopping Mix

of this experiment; Causeway’s overhead on TPC-W’s throughput remains less than 4%, further it does not increase with increasing load on the system and remains fairly constant. This result shows that Causeway may be used in a production environment without any substantial performance degradation.

5.5 Summary

We have designed Causeway, operating system support for facilitating development of meta-applications to control and analyze multi-tier applications. Causeway provides interfaces for metadata injection and access which can be used for propagation of metadata in multi-tier applications. Propagated metadata can be accessed and used to implement the desired meta-application. We have implemented Causeway in the FreeBSD operating system kernel. The complexity of adding transfer points in the FreeBSD kernel for system-visible channels was modest. Causeway’s support for system-visible channels suffices for metadata propagation in an implementation of the TPC-W [Cou02] benchmark using Apache and MySQL — no modification to Apache or MySQL was required. We measured the overhead of Causeway and found it small enough so that it can be used in a production environment. Further, the overhead scales well with increasing metadata size and load on the application. We have
demonstrated the use of Causeway by implementing a multi-tier priority enforcing system and using it to achieve global priority enforcement on our implementation of the TPC-W benchmark. This required adding only about 150 lines of code on top of Causeway. We describe this system in Appendix A.

Causeway has proved useful in our implementation of transactional profiling. Whodunit utilizes Causeway to propagate transaction context information across tiers as Causeway metadata. These two meta-applications provide ample testimony to the utility of the Causeway framework.

5.6 Transactional Profiling Across Distribution

In a distributed environment, stages exchange request and response messages among themselves to execute transactions. We give a mechanism to identify request and response messages, and to establish transaction contexts across such distribution.

We explain the mechanism of establishing transaction contexts in a hypothetical two-stage application using RPC-style (request/response) communication. Let us assume that stage 2 (callee) provides an RPC service routine, and stage 1 (caller) has two transaction paths, one through procedure foo and the other through procedure
bar, that make calls to this RPC service. Figure 5.1 shows the call path trees [ABL97] of the caller and the callee.

For transaction propagation via messages, we define the transaction context at a message send point as being the call path [Hal92, HG93] of the program. The transaction context at the caller (in the above case) at the RPC call through foo is \( c_{foo} \), and for the RPC call through bar it is \( c_{bar} \). Then, two different transaction contexts reach stage 2 (for two different transactions). When stage 2 executes the transaction from foo, its transaction context is \( c_{foo} \) suffixed by its call path. Similarly, when it executes the transaction from bar, its transaction context is \( c_{bar} \) suffixed by its call path. When profile sampling occurs at stage 2, its profile data are labeled by
its transaction context. This allows profile data at stage 2 to be maintained separately
for each different transaction.

When stage 2 sends back a response to stage 1, the call path within stage 2 is
the same — let us denote it by $c_{\text{response}}$ — irrespective of the RPC through $\text{foo}$ or
through $\text{bar}$. This is because the program on stage 2 executes the same sequence
of procedure calls to send the reply back to stage 1. By definition, the transaction
context is then $c_{\text{foo}}#c_{\text{response}}$ or $c_{\text{bar}}#c_{\text{response}}$ ($#$ is a special delimiting character).
This transaction context is then sent to the caller along with the response data.

The caller identifies that a prefix of the callee's transaction context actually origi-
nated from itself. From that the caller infers that the message must be a reply to its
RPC request, and that it does not inherit the callee's transaction context.

Transactional profiling is performed as follows. The caller maintains its own profile
and information about the transaction contexts it sends to the callee. The callee
annotates its profile information with its current transaction context. Post mortem,
the caller's and the callee's profile data are stitched together using the annotations
of the profile. Figure 5.2 is a graphical representation of the resulting transactional
profile for the two different transactions in the above application. Nodes represent
procedures and they have associated profile data (not shown in the figure). The call
path tree of the callee appears twice, each corresponding to two different transaction
contexts (received from the caller). Nodes performing send and receive for request
and response messages are connected with edges labeled request and response during
the post mortem phase.

5.6.1 Implementation

As above, we describe the implementation with respect to an RPC (request and
response) between two stages that send and receive messages between themselves.
On a message send operation, Whodunit computes the transaction context of the sender as the call path in the program leading to the send operation. Then, it creates a synopsis of this transaction context and propagates it to the receiver. A synopsis is a compact and unique representation of a transaction context. Whodunit uses 4 bytes for each transaction context synopsis, and maintains transaction contexts and their synopses in a dictionary. Propagating a synopsis instead of a transaction context reduces Whodunit's communication overhead.

If $\alpha$ is the transaction context at the send node of a caller for an RPC, then $\text{synopsis}(\alpha)$ is its transaction context synopsis which becomes the transaction context at the callee. If $\beta$ is the call path at the send node of the callee for response to the same RPC, then $\alpha\beta$ is its transaction context. $\text{synopsis}(\alpha)\#\text{synopsis}(\beta)$ is its transaction context synopsis, which is formed by concatenating $\text{synopsis}(\alpha), \#,$ and $\text{synopsis}(\beta)$, where $\#$ represents a unique symbol acting as a delimiter between the prefix and the suffix of the transaction context. This representation allows the caller's transaction context to be identified as a prefix in the callee's transaction context (in the response message). Whodunit computes and propagates transaction context synopses without requiring any modification to the distributed application by providing wrappers for the send and receive operations. Within these wrappers Whodunit uses Causeway to propagate the synopses among stages.

On each stage Whodunit manages a dictionary of CCTs labeled by transaction context synopses. At the callee, the received transaction context becomes its transaction context, and Whodunit switches to the CCT labeled with the same transaction context. If such a CCT does not exist, Whodunit creates one, labels it with the transaction context and switches to it. The callee's profile samples are collected in this CCT until the callee switches to another CCT (i.e., when it executes with a new transaction context). The caller also notes the association between its CCT and the
transaction context synopsis when it sends a request to the callee. This is required because on receiving the response back from the callee, it needs to switch back to the CCT from which the request originated (since it may have changed CCTs in the meanwhile and now it needs to execute on behalf of the earlier transaction that had issued the RPC request).

To summarize, the wrappers for send and receive operations do the following.

**send wrapper:**

1. Compute the transaction context synopsis of the node in the CCT performing the send.

2. Associate the CCT with the above computed transaction context synopsis.

3. Perform the send operation and piggy-back transaction context synopsis on application data (via Causeway).

**receive wrapper:**

1. Perform the receive operation and obtain the piggy-backed transaction context synopsis from the sender (propagated via Causeway).

2. At the callee, switch to the CCT corresponding to the sender's transaction context synopsis. At the caller, identify the prefix of the received transaction context synopsis that originated from itself, and switch to the CCT from which that prefix originated.
Chapter 6

Transaction Crosstalk and Profiling Heterogeneous Layers of Execution

In this chapter we describe the design and implementation of transaction crosstalk, where we measure interference among concurrent transactions, and how we profile applications that have heterogeneous layers of execution — such as, native and interpreted layers.

6.1 Transaction Crosstalk

Concurrent transactions can interfere with each other, slowing their execution. Such interference stems from lock contention. We term such interference as transaction crosstalk. Sometimes, transaction crosstalk may have a pronounced effect on the performance of transactions, e.g., increasing transaction latency significantly. Previous work in the area of distributed profiling has not addressed this issue of crosstalk. Transactional profiling measures and presents transaction crosstalk to pinpoint performance problems due to interfering transactions.

We model transaction crosstalk as follows. Concurrent transactions usually acquire and release locks to access shared state. Locks may be accessed in shared or exclusive mode. When one transaction acquires a lock in an exclusive mode, all other transactions requesting the same lock need to wait. We measure the waiting time for lock acquire operations for all transactions. We also remember the transaction causing the wait. We present these two pieces of information as transaction crosstalk.
To illustrate, assume a two-stage application with a web server front stage followed by a database server. Assume, the application implements two different transaction types $t_A$ and $t_B$. Consider, two concurrent transaction instances $t_{A_i}$ and $t_{B_j}$ executing in the database stage. If $t_{A_i}$ and $t_{B_j}$ both need to acquire a lock $lock_x$ in exclusive mode, one of them (say, $t_{B_j}$) has to wait while the other ($t_{A_i}$) is holding the lock. In this case, transaction crosstalk includes the length of $t_{B_j}$'s wait at the lock acquire operation, and the transaction instance that causes the wait, $t_{A_i}$. When presenting the profile data on transaction crosstalk, we find the average waiting time for all $t_B$ transaction instances that need to wait for $t_A$ transaction instances, and present that as transaction crosstalk for the ordered pair $(t_B, t_A)$ of transactions, i.e., where $t_B$ waits for $t_A$. Similarly, crosstalk for the ordered pair $(t_A, t_B)$ is measured and presented.

Transaction crosstalk must not be confused with measuring lock waiting times in a multithreaded application. For example, the Tmon tool [JFL98] measures and presents lock waiting times in single-stage multithreaded applications. In contrast, transaction crosstalk measures waiting times caused by interfering transactions in a distributed environment. Consider, the two-stage application as described above. If we measure the lock waiting times for the threads in the database server, we have no transaction-level information. That is, we cannot infer what transaction is waiting, and what transaction is causing the wait. By establishing transaction contexts in a multi-tier application, transaction crosstalk presents a more meaningful presentation of lock waiting times at the different stages of the application.

6.1.1 Implementation

Whodunit provides wrappers for lock acquire and release operations to record transaction crosstalk. The magnitude of wait time at lock acquire operations is measured
in the wrapper for the acquire operation. Whodunit maintains a dictionary of lock objects to record transaction contexts that currently have acquired them in exclusive mode. The dictionary is updated when a transaction context acquires and when it subsequently releases a lock. When a transaction waits to acquire a lock, Whodunit looks up the dictionary to find the transaction context holding the lock (and hence causing the wait). Transaction crosstalk data is presented as a two-dimensional matrix of lock wait times for each pair of interfering transaction contexts.

6.2 Transactional Profiling Through Heterogeneous Layers of Execution

Transactional profiling establishes transaction contexts in a program composed of native and interpreted components. To achieve this we augment the machine-level call path with decorations. A decoration is extra information that we associate with a machine-level call path. The machine-level call path consists of the call path through the native component and that through the interpreter engine. In particular, we decorate those segments of the machine-level call path that correspond to the interpreter engine with the source-level call paths of the interpreted program — the remainder of the machine-level call path segments do not get decorated. Under this design, at runtime the profiler collects the machine-level call path profile with the corresponding decorations. Now we describe how a decoration is collected and stored, and how it establishes a transaction context.

To decorate the machine-level call path profile with source-level call paths at runtime, the profiler requires the interpreter engine to provide an interface which it may use to obtain the source-level call path of the interpreted program at any instant. When a program starts, the profiler records where in the process address
space the interpreter engine is loaded and its extent. On a profile sample, the profiler inspects the machine-level stack to find if any of its segments belong to the interpreter engine. If such a segment exists in the native stack, the profiler queries the interpreter engine's decoration interface to obtain the source-level call path of the program being interpreted. The profiler then records this decoration along with the machine-level stack in the CCT. A CCT node is extended to store the decoration information. Thus, the CCT now contains machine-level call paths (possibly) augmented by decorations.

Figure 6.1 illustrates the above concept. The call path on the left hand side is the machine (native) call path at a profile sample, where segment $Y$ belongs to the interpreter engine. Assume $Z$ is the source-level call path in the interpreted program at that instant. Then replacing $Y$ by $Z$ in the native call path generates the source level call path profile sample, as shown in the call path on the right hand side of Figure 6.1. All CCT nodes in the call path segment $Y$ are annotated with label $Z$ when the above profile sample is collected. The CCT, thus, maintains the source-level profile of the program. The call path obtained by replacing the native components belonging to the interpreted engine by the source-level call path of the interpreted program is output during presentation of the profile data.
When two call paths are compared, their machine-level call path values as well as their decorations are compared. If either one does not match, then the comparison fails. Thus now, two identical machine-level call paths may not match against each other. For example, if the profiler profiles two different interpreted programs that are executed by the same interpreter engine, it may record the same machine-level call path for the two programs but their decorations differ enabling the profiler to present different transaction contexts for these two programs. A transaction context, at any instant in program execution, is constructed from the machine-level call path by replacing those segments having decorations with their corresponding decorations. Thus, machine-level call path segments corresponding to the interpreter engine get replaced by their corresponding source-level call paths in the interpreted program.

6.2.1 Implementation

Our current implementation of Whodunit supports transactional profile of heterogeneous programs having Java or Python as their interpreted component. Java and Python have support in their run-time systems to provide the source-level call path of programs written using these languages.

Java

The Java Native Interface (JNI) allows a native program to call a Java program or vice versa. Using JNI, a profiler written in C can be easily integrated with a program having a Java component. The Java programming language exports the Java Virtual Machine (JVM) Tool Interface (JVTI) [JVM]. JVTI is intended to be used by profiling tools, among others, that need access to the state of the Java application in the JVM. JVTI exports an interface to obtain the Java Virtual Machine stack frame of a thread running in the JVM. This interface is used by Whodunit to decorate
the machine-level call path profile of a program with a Java component.

Python

The Python/C API [vR05] provides a mechanism to write extensions to the Python interpreter using C or C++, or to embed Python scripts in C/C++ programs. This API provides a callback mechanism enabling profiling and tracing of a Python program from a C/C++ program. The callback mechanism allows a profiling tool to inspect the Python interpreter's source level stack frame. Similar to the JVMTI interface for Java, the Python/C API interface is used by Whodunit to decorate the machine-level call path profile of a program that has a Python component.
Chapter 7

Case Studies

We demonstrate the use of Whodunit in obtaining the transactional profile of a few applications. We also measure Whodunit’s overhead on the performance of these applications. The machines used in our experiments have a 2.4 GHz Pentium Xeon CPU, 2 Gigabytes of memory, and are connected by switched Gigabit ethernet.

7.1 Apache and MySQL

An important aspect of transactional profiling is detection of transaction flow through shared memory and propagation of transaction contexts across such transaction flow. We have described our algorithm for transaction flow detection and the mechanism for passing transaction context in Chapter 3. An important test of this algorithm is to demonstrate its use in real applications with concurrent threads and shared state.

We choose two popular multi-threaded servers — Apache (version 2.0.54) and MySQL (version 4.0.25). We profile executions of these servers while executing traces of web and database workloads. Our algorithm detects no transaction flow in MySQL. Whodunit detects a shared counter in MySQL, but correctly deduces that it does not constitute transaction flow.

For Apache, our algorithm detects transaction flows through shared memory and establishes transaction contexts across them. Figure 7.1 shows a portion of the transactional profile of Apache under a web workload trace obtained from our department’s web server at Rice University. Nodes represent procedure names, solid edges represent
Figure 7.1: Transactional profile of Apache under the Rice web workload.

procedure calls, dashed edges represent transaction contexts established by Whodunit, and triangles show the percentage of Apache's profile collected by a procedure and its children.

Figure 7.1 shows flow of transaction contexts from a listener thread to multiple worker threads. The listener thread accepts incoming connections and puts them in a shared queue using the \texttt{ap\_queue\_push} routine, while the worker thread dequeues them via the \texttt{ap\_queue\_pop} routine. The shared queue is protected by a mutex. Whodunit successfully detects this transaction flow. Whodunit also detects a synchronized memory allocator in Apache, but it does not satisfy the rules of transaction flow.
7.1.1 Overhead of Emulation

First, we measure Whodunit’s performance overhead on Apache (version 2.0.54) due to emulation of its critical section. Then we measure the absolute cost of emulating Apache’s critical sections as shown in Figure 3.1.

A transaction flows through shared memory in Apache only when a new connection is accepted by the listener thread and put in a shared queue to be later picked up (and processed) by a worker thread. Obviously, if all connections are persistent and no new connections are established, Whodunit does not need to emulate any code, and as such the application can proceed in “direct execution” mode without any overhead. In our experiment, we model a more realistic scenario, in which we execute a real web workload trace (collected at the web server of the Computer Science Department, Rice University) against Apache while profiling it using Whodunit. The workload simulates concurrent clients that open new connections, send a few HTTP requests over them, close the connections, and then again send more HTTP requests over new connections. Thus, Whodunit has to repeatedly emulate the critical sections.

For the above workload, we compare the throughput of Apache for two different cases: while not being profiled ("normal execution") and while being profiled with Whodunit. For the normal execution mode, Apache’s peak throughput for the above workload is 393.64 Mb/s. While being profiled with Whodunit, Apache’s peak throughput is 384.58 Mb/s, an overhead of only 2.3%. Whodunit’s overhead is small because it uses QEMU [Bel], an efficient CPU emulator. QEMU caches the translated machine instructions of the emulated program, thereby obviating the need for repeated translations of Apache’s critical sections.

Finally, we construct a micro-benchmark to measure the absolute cost of emulation in machine cycles. We measure the cost of executing the critical sections of
Apache under direct execution mode and under emulation using Whodunit. Table 7.1 shows the results of this experiment. `ap_queue_push` and `ap_queue_pop` are the two critical sections of Apache. They take about 132 and 110 machines cycles to execute in direct execution mode, respectively. They take more than 62K and 40K cycles, respectively, when they are translated to intermediate code and then the intermediate code is executed. Due to QEMU’s caching, translation is performed only once, and subsequently when the critical sections are emulated, the translated code from the cache is executed. The cost of emulation for the two critical sections then drops to about 11K and 12K cycles, respectively.

<table>
<thead>
<tr>
<th>Critical Section</th>
<th>Direct Execution</th>
<th>Translation and Emulation</th>
<th>Emulation only</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ap_queue_push</code></td>
<td>131.64</td>
<td>62508</td>
<td>11606.8</td>
</tr>
<tr>
<td><code>ap_queue_pop</code></td>
<td>109.72</td>
<td>40852</td>
<td>12118</td>
</tr>
</tbody>
</table>

Table 7.1: Execution time of Apache’s critical sections (in machine cycles) for the different modes of execution.

7.2 Squid Web Proxy Cache

Squid [SQU] is an event-driven, open-source proxy caching server. Squid does not use an event-library like `libevent` [Pro03]. So, we modify the Squid program to support transactional profiling. The modifications are limited to Squid’s event loop (`comm_poll`) and the procedure to register interest for events (`commSetSelect`). These modifications required about 120 lines of code, while Squid contains more than 125,000 lines of code.

Using Whodunit on our modified version of Squid we are able to obtain its transac-
Figure 7.2: Transactional profile of Squid under the Rice web workload.

tional profile. We use Squid version 2.5.STABLE12 for our experiments. We execute a web workload trace using a client emulator program against Squid while profiling it. This is the same workload that is used for the Apache experiments. Squid, the origin server and the client program all run on separate machines.

Squid’s main event handlers are httpAccept, which accepts incoming connection requests, clientReadRequest, which reads the client request, commConnectHandle, which opens a connection with the origin server, httpReadReply, which receives content from the origin server, and commHandleWrite, which sends a reply to the client. The transactional profile of Squid for the above experiment is shown in Figure 7.2. In this figure, a dashed edge represents a transaction context established by a sequence of event handlers. The event handler commHandleWrite appears in two transaction contexts: once after the sequence httpAccept-clientReadRequest, and once
after httpAccept-clientReadRequest-httpReadReply. Of these two sequences of
event handlers, the former corresponds to when the request is found in the proxy
cache (cache hit), while the latter corresponds to a cache miss when the origin server
needs to be contacted. Whodunit enables us to distinguish the time spent in the
commHandleWrite event-handler for cache misses and cache hits, a distinction which
is not provided by a regular profiler.

7.2.1 Whodunit's Overhead on Squid

We measure the peak throughput delivered by Squid under two conditions: while
being profiled by Whodunit, and while running with profiling disabled. For the web
trace workload (from the CS department at Rice University) Squid's peak throughput
with profiling disabled is 262.27 Mb/s which drops to 247.85 Mb/s while being profiled
— an overhead of about 5.5%.

7.3 Haboob

Haboob is a SEDA-based web server [WCB01]. Using Whodunit, we are able to profile
the different transactions of Haboob. We subject Haboob to the same web workload
that we used for Apache and Squid. We run Haboob and the client program on
different machines.

Figure 7.3 shows the transactional profile of Haboob under the above workload.
Nodes represent stages (labeled with stage names) and edges represent transaction
flow between stages. The ListenStage listens on the server socket, the HttpServer
stage accepts client connections, the ReadStage reads packets from the client, the
HttpRecv stages parses the client Request, the CacheStage implements an internal
cache, the MissStage handles cache misses and schedules disk reads via the File I/O
Figure 7.3: Transactional profile of Haboob under the Rice web workload.

Stage, and the WriteStage sends the response back to the client. Figure 7.3 shows that two different transaction types occur in Haboob: a transaction may flow from the CacheStage to the WriteStage either via the cache hit path or via the cache miss path. For each transaction context, the total percentage of CPU profile spent in a stage is shown in a triangle by the node representing a stage. For example, the figure shows that 37.65% of Haboob’s total CPU use occurs in the WriteStage reached via the cache hit path and 46.58% in the same stage reached via the cache miss path. Whodunit measures the time used in WriteStage for the two different transaction paths. This information is not provided by a regular profiler.
7.3.1 Whodunit's Overhead on Haboob

We measured the peak throughput delivered by Haboob under two conditions: while being profiled by Whodunit, and while running with profiling disabled. For the web trace workload (from the CS department at Rice University) Whodunit reduces Haboob's peak throughput by about 4.2%. Peak throughput drops from 31.16 Mb/s with profiling disabled to 29.84 Mb/s while being profiled.

7.4 TPC-W

TPC-W [Cou02] models an online bookstore. It defines a set of fourteen different transactions, e.g., searching and buying of books. Our TPC-W implementation [dyn] is composed of Java servlets for the TPC-W transactions, and the MySQL database server (version 4.0.25) to store its persistent data. We use the Apache Tomcat servlet container (version 4.1.31) as the web server/application server executing the servlets. The fourteen different TPC-W transactions are implemented as fourteen different Java servlets. TPC-W serves dynamic as well as static content, e.g., thumbnail and full images of books. In our setup Squid [SQU] executes in front of Tomcat to cache TPC-W's static content. All TPC-W requests flow through Squid to Tomcat and MySQL. In our setup Squid, Tomcat and MySQL run on separate machines.

We subject the TPC-W implementation to a browsing mix workload [Cou02] from multiple, concurrent clients and profile it using Whodunit. For each TPC-W transaction page, that is dynamically generated by a servlet, Squid sends a request to Tomcat via the same call path, and hence, transfers the same transaction context to Tomcat. Since each TPC-W interaction is implemented as a separate Java servlet, Whodunit extends a separate transaction context from Tomcat to MySQL for each interaction. Thus, for each TPC-W interaction, a separate transaction context is established in
<table>
<thead>
<tr>
<th>Transaction</th>
<th>MySQL CPU profile (%)</th>
<th>Mean crosstalk wait time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdminConfirm</td>
<td>0.82</td>
<td>93.76</td>
</tr>
<tr>
<td>AdminRequest</td>
<td>0.00</td>
<td>6.68</td>
</tr>
<tr>
<td>BestSellers</td>
<td>51.50</td>
<td>22.16</td>
</tr>
<tr>
<td>BuyConfirm</td>
<td>0.04</td>
<td>68.55</td>
</tr>
<tr>
<td>BuyRequest</td>
<td>0.03</td>
<td>0.11</td>
</tr>
<tr>
<td>CustomerRegistration</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Home</td>
<td>0.57</td>
<td>1.51</td>
</tr>
<tr>
<td>NewProducts</td>
<td>3.29</td>
<td>1.59</td>
</tr>
<tr>
<td>OrderDisplay</td>
<td>0.01</td>
<td>0.09</td>
</tr>
<tr>
<td>ProductDetail</td>
<td>0.22</td>
<td>0.66</td>
</tr>
<tr>
<td>SearchRequest</td>
<td>0.16</td>
<td>1.15</td>
</tr>
<tr>
<td>SearchResult</td>
<td>43.28</td>
<td>5.52</td>
</tr>
<tr>
<td>ShoppingCart</td>
<td>0.07</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 7.2: MySQL CPU profile (%) and mean crosstalk waiting times for the different TPC-W transactions for browsing mix workload with 100 concurrent clients.

MySQL. For each transaction, Whodunit maintains its resource usage separately at both Tomcat and MySQL. Such separation of resource utilization at MySQL would not have been possible by using a conventional profiler, e.g., gprof [GKM82]. This separation is made possible by Whodunit’s use of call path profiling at Tomcat (each TPC-W transaction is executed by a separate servlet and hence has a distinct call path), and by tracking transaction flow across RPCs between stages (Tomcat makes database RPC calls into MySQL).

Whodunit shows that the average resource usage at Tomcat by the different TPC-W transactions is roughly the same. Table 7.2 shows the summary of MySQL’s profile
<table>
<thead>
<tr>
<th>Transaction</th>
<th>AC</th>
<th>NP</th>
<th>AR</th>
<th>OD</th>
<th>BS</th>
<th>OI</th>
<th>BC</th>
<th>PD</th>
<th>BR</th>
<th>SRq</th>
<th>CR</th>
<th>SRs</th>
<th>H</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdminConfirm</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>28.2</td>
<td>-</td>
<td>-</td>
<td>38.0</td>
<td>-</td>
<td>4.7</td>
<td>-</td>
<td>11.0</td>
<td>11.8</td>
<td>-</td>
</tr>
<tr>
<td>BuyConfirm</td>
<td>-</td>
<td>16.2</td>
<td>-</td>
<td>-</td>
<td>18.0</td>
<td>-</td>
<td>-</td>
<td>5.9</td>
<td>-</td>
<td>5.2</td>
<td>-</td>
<td>4.9</td>
<td>18.2</td>
<td>-</td>
</tr>
<tr>
<td>BestSellers</td>
<td>0.1</td>
<td>0.1</td>
<td>-</td>
<td>19.3</td>
<td>0.2</td>
<td>-</td>
<td>0.3</td>
<td>0.6</td>
<td>0.3</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.4</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 7.3: Transaction crosstalk matrix (lock wait time in milliseconds) for AdminConfirm, BuyConfirm, and BestSellers transactions for browsing mix workload with 100 concurrent clients.

Information as measured by Whodunit while executing the workload from 100 concurrent clients. This table shows the percentage of MySQL CPU profile (%) per transaction, and the mean crosstalk waiting time (in milliseconds) per transaction. AdminConfirm's crosstalk waiting time caused by waiting for locks held by other transactions is about 94 milliseconds on average — the maximum among all transactions. Table 7.2 further shows that about 51% and 43% of MySQL's total CPU time is spent in response to queries for the BestSellers and the SearchResult interactions, respectively. Thus, the profile output of Whodunit suggests that the interface between the servlets for the AdminConfirm, BestSellers, and SearchResult interactions and the database server, i.e., the database queries, are the candidates for possible optimizations.

Table 7.3 shows the details of transaction crosstalk profile data for the AdminConfirm, BuyConfirm, and BestSellers transactions. For each of the TPC-W transactions, the lock wait times (in milliseconds) incurred by these three transactions is shown. In this table AC stands for AdminConfirm, NP for NewProducts, AR for AdminRequest, OD for OrderDisplay, BS for BestSellers, OI for OrderInquiry, BC for BuyConfirm, PD for ProductDetail, BR for BuyRequest, SRq for SearchRequest, CR for CustomerRegistration, SRs for SearchResult, H for Home, and SC for ShoppingCart. Table 7.3 illustrates how transaction crosstalk profile data is presented for
a multi-tier application.

AdminConfirm issues a heavy-weight database query involving sorting of table records, creation of a temporary table, and updating a single row in the item table. The item table is also accessed by read-only SQL queries from most of the other transactions. Since AdminConfirm requires an exclusive lock on item table, its crosstalk wait time is high.

However, AdminConfirm need not acquire an exclusive (write) lock on the entire table, but rather an exclusive lock on the row being updated. The item table type in our database is a MyISAM table. The MyISAM table type supports only table-wide locking. The InnoDB table type supports locking on rows. We convert the item table to an InnoDB type. With this optimization we execute a browsing mix workload on TPC-W with varying number of concurrent clients. We measure the average response time of the AdminConfirm transaction and compare it with the same for the original TPC-W system that uses a MyISAM type item table. Figure 7.4 shows the results. Using the above optimization, we achieve between 9% and 72% reduction in AdminConfirm’s average response time. For example, under the browsing mix workload of 100 concurrent clients, AdminConfirm’s response time decreases from 640 milliseconds to 550 milliseconds.

The servlets for the BestSellers and the SearchResult interactions both issue heavy-weight read-only SQL queries performing sorting on database records. Clause 6.3.3.1 of the TPC-W specification [Cou02] states that the results of the BestSellers and the SearchResult transactions (involving search by subject) may be cached for a duration of 30 seconds. Further, the results for SearchResult interactions involving search by title or author may be cached forever. These optimizations have not been performed in the implementation of TPC-W that we are using.

We add caching of the BestSellers and the SearchResult interaction results in the
Figure 7.4: Average response time for AdminConfirm, BestSellers & SearchResult transactions under the original and the optimized cases.

Java servlets. Caching substantially reduces the mean response time for these two transactions as shown in Figure 7.4. Figure 7.5 shows TPC-W's throughput with and without caching of the BestSellers and the SearchResult transactions. Without caching, the database CPU becomes the bottleneck with a fewer number of clients (about 200 as shown in Figure 7.5). However, with caching, throughput increases almost linearly with an increasing number of clients up to about 450 clients, at which time the database CPU becomes the bottleneck and throughput does not increase any further. The peak throughput obtained under caching is 3376 interactions/minute — close to three times the peak throughput of 1184 interactions/minute obtained under
Figure 7.5: Throughput (in transactions/minute) under browsing mix workload with and without caching.

the original (no caching) case.

7.4.1 Whodunit’s Overhead for TPC-W

We measure the peak throughput of TPC-W under four scenarios: no profiling, profiling using csprof, profiling using Whodunit, and profiling using gprof. We used the same sampling frequency for csprof, Whodunit and gprof — equal to gprof’s default sampling frequency (666 times per second) on our platform.

Table 7.4 shows the result of the above experiment. For TPC-W, csprof’s overhead (less than 3%) is much less than gprof’s overhead (about 24%), although both
Table 7.4: Peak throughput (transactions/minute) of TPC-W under various profiling tools.

<table>
<thead>
<tr>
<th>No profile</th>
<th>Profile with cprof</th>
<th>Profile with Whodunit</th>
<th>Profile with gprof</th>
</tr>
</thead>
<tbody>
<tr>
<td>1184</td>
<td>1151</td>
<td>1150</td>
<td>898</td>
</tr>
</tbody>
</table>

sample at the same frequency. gprof inserts instrumentation code to count procedure calls. As a result, its overhead is proportional to the number of calls executed by a program and higher than cprof's overhead, which remains relatively constant regardless of the number of calls [FMCF05]. Whodunit's overhead on top of cprof is less than 0.1%. This additional overhead of Whodunit comes from propagating transaction contexts among stages and managing the profile data of a stage based on transaction contexts. At peak throughput, 92.52MB of data and 0.95MB of transaction context is transferred among the stages of TPC-W — a communication overhead of about 1%.

7.5 tBoard

We describe our experience with using Whodunit to profile and optimize tBoard [tbo], an open source, Python powered bulletin board framework. tBoard defines the following set of transactions: display stories of the day (Display), display a particular story and its comments (Story), search and display stories and comments containing a given list of keywords (Search), register a new user (UserReg), post a story or a comment (Post), and some administrative transactions *.

*Shorthand name for a transaction is shown within a pair of parentheses following the transaction description.
tBoard is composed of a web/application server and a database server. The various functionalities of the tBoard transactions are implemented in Python scripts. Python [pyt] is an interpreted, object-oriented language. We used the Apache web server built with the mod_python module [mod] as our web/application server, and MySQL as our database server. The web server (Apache) and the application server (Python interpreter) run in the same address space. Apache (native) and the tBoard Python scripts (interpreted) constitute a heterogeneous tier in the application. Our software environment consists of the following: Apache version 2.0.54, mod_python version 3.2.8, Python version 2.4.2, and MySQL version 4.0.25. In our hardware setting, Apache and MySQL run on two separate Pentium 4 Xeon (2.4GHz, 2GB memory) machines connected by switched Gigabit ethernet.

The database mainly consists of users, stories, and comments. Our database contains 500,000 users, 6000 stories, and about 213,000 comments. This database simulates the total data for about 300 days of a bulletin board that receives about 20 stories per day and about 35 comments per story. We generate the story and comment bodies using words from a dictionary. The length of a story or a comment is less than 8KB. Short stories and comments are more common, so we use a Zipf-like distribution for story and comment length [BCF+90]. We used a workload generator that emulates concurrent clients. About 90% of the workload is read-only, similar to Slashdot’s workload where most accesses are reads [sla].

On executing this workload on tBoard we observed that its peak throughput is about 51 transactions/minute, the database CPU being the bottleneck. We used Whodunit to profile tBoard. Whodunit can be used in two modes: first, where it does not track the transaction context through an interpreted layer in the software stack, and second where it does so. Under the first mode, we were able to propagate the machine level transaction context from Apache/Python to MySQL. This ma-
<table>
<thead>
<tr>
<th>Transaction</th>
<th>MySQL CPU profile (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Story</td>
<td>58.27%</td>
</tr>
<tr>
<td>Post</td>
<td>0.02%</td>
</tr>
<tr>
<td>Search</td>
<td>40.84%</td>
</tr>
<tr>
<td>Display</td>
<td>0.85%</td>
</tr>
<tr>
<td>UserReg</td>
<td>0.001%</td>
</tr>
</tbody>
</table>

Table 7.5: MySQL CPU profile (%) for the different tBoard transactions for a workload with 10 concurrent clients.

Chinese level transaction context consists of the call path through Apache, mod_python, and the Python interpreter. However, the different Python scripts are executed by the same native code, and thus the native level transaction context fails to identify the application level transaction context as it contains an interpreted layer. Using Whodunit in this mode did not identify which tBoard transactions were creating the database bottleneck.

Using Whodunit under the second mode, we were able to extend the application-level transaction context — comprising of call path through the Python interpreted scripts — from Apache/Python to MySQL. Since different Python scripts implement different tBoard transactions, a separate transaction context for each of the tBoard transactions can be established from Apache/Python to MySQL. Thus, the CPU profile at MySQL for each of the different tBoard transactions can be maintained separately. Table 7.5 shows the percentages of MySQL CPU profile for the tBoard transactions as presented by Whodunit. About 99% of MySQL's CPU profile is due to two transactions — about 58% from the Story transaction, and about 41% from the Search transaction. Inspection of the workload revealed that about 32% and 7% of the transactions were of type Story and Search respectively.
Figure 7.6: tBoard throughput (in transactions/minute) with and without caching

We found that these two transactions issue heavy-weight SQL queries comprising of join and sort operations. However, these queries do not perform any writes to the database. One solution to reduce the load on the database server is to cache the results of these transactions in the application server. We modified tBoard to add caching support for these two transactions at the application server. We compare the performance of the original, unmodified tBoard application to that of the optimized version with caching support for Story and Search transactions. Figure 7.6 compares the throughput of these two versions with increasing number of concurrent clients. Using caching, tBoard’s peak throughput increases by almost a factor of two — from about 51 transactions/minute to about 91 transactions/minute.
Chapter 8

Related Work

In this chapter I provide a survey of profiling models and tools for distributed systems and compare them with transactional profiling. These various pieces of related work have been performed by the research community and the industry.

8.1 Research Tools

Pip [RKW+06] is a system to compare actual behavior of a distributed system to its expected behavior in order to identify correctness and performance bugs in the application. Pip allows programmers to express the structure of a distributed application in a declarative language. Such structure of an application consists of communication channels, timing and resource consumption. An example of such a structure can be over-usage or under-usage of some resource. Over-usage of resource may indicate performance bugs. Under-usage of a resource may indicate truncated execution as a result of a correctness bug.

Pip [RKW+06] expects the programmer to express the structure of an application using annotations. This approach is more invasive than transactional profiling. Further, it may not always be possible to characterize the "unexpected" behavior of an application. This is because, for example, the behavior of an application may vary widely depending on the workload that it executes. While the expected behavior of the application may be inferred on the basis of a moderately loaded system, the observed behavior may change drastically under higher load and Pip will character-
ize that behavior as unexpected. The goal of transactional profiling is to associate resource usage with transaction execution and not to identify correctness bugs. As a result, transactional profiling does not need to characterize expected and unexpected behavior of an application. Further, transactional profiling can show the differences among the resource usage characteristics of the application executing under different levels of workload.

Stardust [TSW+06] is a system to track execution of requests in a distributed storage system. Stardust uses an instrumentation framework that generates log records containing information about activities associated with the execution of requests. Such activities consist of events related to CPU demand, network demand, buffer cache usage, and disk demand. The instrumentation framework tags each record with a unique identifier corresponding to the executing request. This identifier is used to relate activities on different nodes of the distributed storage system. Stardust exports a querying mechanism to visualize the resource usage by different requests and the causality of different components in the system.

Stardust [TSW+06] tackles the problem of performance characterization in a distributed storage system. Stardust’s mechanism is highly intrusive because it uses an instrumentation framework to insert identifiers along request execution paths — programmers must provide the necessary instrumentation in their code. Transactional profile, on the other hand, is less intrusive because most of its instrumentation is internal to system libraries and the kernel. Further, the mechanisms of transactional profiling are applicable to any generic distributed system, including distributed storage systems. Finally, Stardust is not generic enough to capture event based concurrency model and shared memory communication that are handled in transactional profiling.

Project 5 [AMW+03, RWM+06] infers causal paths from message traces in a dis-
tributed application to identify bottleneck components. Aguilera et al. [AMW⁺03] addressed this problem in a local area network; Reynolds et al. [RWM⁺06] provided a solution for the same problem for distributed systems over a wide area network. Project 5 is the least invasive to the application as it only captures the message used for communication among nodes and analyzes them to infer causal information. While it can provide information such as locating the bottleneck node in a distributed application, unlike transactional profiling it cannot provide profile statistics on procedures in that component. Further, often different transactions execute through the same set of nodes in a distributed application. In that case, Project 5 cannot infer bottleneck causing transaction and corresponding execution path in those nodes. Finally, Project 5 does not track transaction execution through shared memory and transaction crosstalk due to interfering transactions.

Magpie [BDIM04, IBB⁺04] measures per-transaction resource consumption in a distributed application. Magpie makes use of the Event Tracing facilities on Windows [Mic02] that logs transaction execution events (such as message passing). It analyzes the event log on the tiers of the application, and correlates them using an application-specific event schema. Such an event-schema can only be provided by an expert. Also, Magpie is heavily dependent on the operating system to provide a log of transaction execution events — such support is currently available only on Windows. Transactional profiling can track transaction execution in a distributed application without such operating system support or application specific knowledge. Also, Magpie does not track executions through events, stages or through shared memory, and does not measure interference among transactions.

Several performance analysis tools for parallel programs collect call path data, e.g., CATCH [DW02], mpiP [VM01], Paradyn [BM04], and Tau [MSB⁺04]. These tools collect call paths within each process of a parallel program primarily to associate
costs incurred by communication library routines to the program contexts in which the routines are called. Unlike transactional profiling, they do not consider interprocess call-path relationships.

TraceBack [ASM+05] reconstructs control flow in a multi-tier application from the execution history to identify program problems causing faults, viz., crashes, hangs or exceptions. While both can handle mixed-mode applications, transactional profiling and TraceBack perform two different tasks — performance analysis and first fault diagnosis, respectively — on a multi-tier application. TraceBack instruments a given program to record program execution history. Transactional profiling may use statistical sampling to profile a program. TraceBack, however, must use instrumentation as a fault can occur anytime, and TraceBack must be able to locate its cause.

8.2 Commercial Tools

A few commercial tools to profile distributed or multi-tier applications exist in the market. Because of the feasibility of implementation and economic viability, most of these tools profile distributed J2EE [Sunc] applications, e.g., PerformaSure by Quest [QSa], Optibench from Performant [Per], and Borland ServerTrace [Cor]. For most of these tools, the Java Virtual Machine (JVM) provides a common platform to instrument or analyze any Java/J2EE application.

PerformaSure [QSa] is a tool to identify performance bottlenecks in a distributed J2EE system. It can follow individual transactions through the different tiers of a distributed J2EE system, viz., web server, Java Server Pages (JSP) [Sunf], Enterprise Java Beans (EJB) [Suna], and database server. It uses the Tag-and-Follow (TM) technology to track transaction execution. Under this technology, a transaction is tagged and the tag flows along with the transaction as it is executed by the different
tiers of the system. Tag-and-Follow is made possible by the fact that the application uses well-conditioned communication channels, e.g., Java Remote Method Invocation (RMI) [Sun] for communication among the different tiers of the application. Quest [Que] recommends the following method to optimize distributed J2EE applications. First, use PerformaSure to identify problematic transactions and bottleneck tiers for those transactions. Then, use JProbe [Qsb] for deep inspection within those tiers. JProbe provides method-level call graph profile of a Java application or component. It links resource usage by a program to the lines of its source code, thereby identifying program components or methods that may require optimization.

While the tools from Quest (as mentioned above) require no instrumentation, Optibench [Per] employs instrumentation of web servers (through plug-ins) and bytecode instrumentation of J2EE applications to capture execution events of transactions and associated resource usages. This information is stored in a repository and correlation is applied on these events to construct transactions. Optibench also has the capability to replay these transactions later to re-create the workload or aid in debugging.

Borland ServerTrace [Cor] is a tool similar to PerformaSure. It can analyze performance of a distributed J2EE application using its "Track and Trace" technology. ServerTrace can provide breakdown of resource usage by transactions in different components of the application, viz., JDBC [Sunb], JSP, EJB, JNDI [Sund], JMS, etc. Then, one can drill down further to look at resource usage of each component. ServerTrace can relate resource usage to the source code of the program.

These commercial tools can only be applied to J2EE applications, whereas transactional profiling can be applied to any general distributed application. Also, these commercial tools do not track transactions through shared memory or through event or stage-based tiers unlike transactional profiling.
Chapter 9

Conclusions

In this dissertation we propose transactional profiling, a novel profiling model for multi-tier applications. Existing profiling techniques are not general enough to track and profile transactions in multi-tier applications. Transactional profiling tracks transactions in a multi-tier application by establishing causal connections among the stages of the application, and by collecting and associating profile data with each individual transaction. Such causal information connecting stages is crucial to identifying requests that cause performance bottlenecks. The objective of this work has been to define the transactional profiling model, design and implement a transactional profiler, and to demonstrate its use on multi-tier applications.

Working toward the above objective, the following have been the results of the work leading to this dissertation:

- We have defined the transactional profiling model. Transactional profiling establishes causal contexts among stages in a multi-tier application, where a stage is a process, a thread, an event-handler or a stage thread (for a SEDA-based application). We have illustrated the profiling model vis-a-vis the call path profiling model, the state of the art in profiling stand-alone applications. Transactional profiling, essentially, extends the call path profiling model to a distributed environment.

- We have designed and implemented Whodunit, a prototype transactional profiler. Whodunit tracks transaction flows across shared memory communication,
across events or stages, across heterogeneous layers of execution, or via inter-process communication by message passing. We describe how Whodunit collects profile data and associates them with transactions in an application. We also describe the design and implementation of transaction crosstalk, an integral part of transactional profiling. Whodunit measures transaction crosstalk, the interference among concurrent transactions, in a multi-tier application.

- We have given a novel algorithm to detect transaction flows through shared memory. Detecting such transaction flows is particularly challenging as shared memory communication is not usually visible to the operating system, unlike message passing via sockets or pipes.

- We have demonstrated the use of Whodunit on web servers, web proxy caches, and dynamic content web sites. These applications use a variety of concurrency models and communication styles, and are representative of a vast majority of multi-tier applications. Using Whodunit we were able to characterize and optimize the performance of a bookstore application and a bulletin board application. Whodunit-inspired optimizations increased the peak throughput of the bookstore and the bulletin board by almost 3x and 2x respectively.

- We have measured the overhead of Whodunit on the profiled application’s performance. We found its overhead to be small — less than 6% — on the various applications that we used. Whodunit’s small overhead makes it applicable to be used in a production setting.
9.1 Future Research Directions

In this dissertation we introduced the concept of transaction crosstalk to measure the interference among concurrent transactions waiting to acquire locks. Transaction context extends the notion of lock waiting times from a multi-threaded program to a distributed environment. Though in transaction crosstalk we measure the wait times for lock acquire operations only, in general, its underlying principle may be applied to measure the wait time for any generic resource. For example, in a distributed program it is not uncommon that concurrent clients of a service interfere while making requests to a common server. In this case, the wait times incurred in obtaining the service could be measured and presented as transaction crosstalk.

The key principle in transactional profiling is tracking transaction flows and extending transaction contexts across the tiers of execution. Though the scope of this work has been profiling only, this principle may be applied to construct a transactional debugger. In such a debugger, one can track the flow of transactions from one tier to another, establish the transaction contexts across the tiers, and insert breakpoints on the different tiers for a specific set of transactions. A transactional debugger would be immensely helpful to solve the correctness issues in a multi-tier application.

Prior art in debugging of distributed systems has looked at providing debugging functionality for remote procedure calls (RPC) only [Coo87, Coo88, Uni]. Establishing transaction contexts in such debugging frameworks will improve their functionality by allowing the user to debug invocations of an RPC on specific transaction contexts, and not just any invocation of the RPC. Further, establishing transaction contexts promises to be useful for debugging multi-tier applications where transactions flow through shared memory, events, or SEDA [WCB01] stage queues — another improvement over current state-of-the-art in debugging of multi-tier applications.
Appendix A

Appendix on Causeway

A.1 Example Use of Causeway: Multi-tier Priority Propagation

Meta-applications to control and analyze the execution of applications can be built easily using Causeway. Transactional profiling is one such meta-application. We illustrate another such meta-application here.

Using Causeway we could rapidly implement a priority propagation system, enabling a multi-tier application to prioritize the execution of requests. Under this system, upon receiving a request the application injects a priority as metadata, Causeway propagates this priority metadata with the execution of the request to each of the stages, and the meta-application uses the priority metadata to enforce priority scheduling on each stage. The meta-application is automatically invoked on each stage by Causeway’s transfer point callback mechanism.

The implementation of the multi-tier priority propagation system on top of Causeway required writing about 150 lines of code. We tested the multi-tier priority propagation system with an implementation of the TPC-W benchmark [Cou02]. No modifications were required in the TPC-W application code, other than the injection of priority metadata.
A.1.1 Metadata Access

The priorities are injected into the system when a request arrives, using the metadata access API of Causeway. We register transfer point callback methods at the transfer points from a kernel thread to a user thread, and from a socket to a kernel thread. These callback methods change the priorities of the user thread and the kernel thread respectively. The first callback method affects the scheduling of MySQL pthreads while the second one achieves the same for Apache processes.

A.1.2 Application

TPC-W simulates an online bookstore. Its implementation consists of a front-end web server, providing an HTTP front-end and serving static content, a middle-tier application server that implements the business logic, and a back-end database server that stores the dynamic content of the site. The benchmark defines 14 interactions with the web site, 13 of which access the database. 6 interactions write to the database, while the others are read-only. Our hardware and software platforms are the same as described earlier in Section 5.4.

A.1.3 Experiment

The goal of the experiment is to demonstrate that multi-tier priority propagation using Causeway, without application modification, has considerable benefits. Our performance metric is the response time of the high-priority requests. We show that the response time of high priority requests is relatively independent of the load imposed on the system. We also demonstrate that enforcing priority at both stage (web server and database server) is superior to only enforcing it at the first stage.

We define a foreground load as a sequence of 100 instances of each TPC-W inter-
action, spaced out in time by one second. We define a background load that directs a steady stream of read-only requests at the site. The background load simulates visitors browsing the web site, while the foreground load simulates customers performing the actions that may lead to purchases at the site, thereby deserving higher priority. We use two different levels of background load: one which overloads the system and one which imposes a moderate load without, however, saturating the system.

We have two levels of priority in the system: a default priority and a high priority. Requests originating from the background load are always tagged with metadata indicating the default priority. To demonstrate the effect of priorities, we perform two experiments, with requests from the foreground load tagged with metadata either indicating the high priority or the default priority. In addition, to demonstrate the difference between single-tier and multi-tier priority enforcement, we run an experiment in which on the web server the priorities are enforced by the transfer point callback methods as described above, but on the database server they are ignored.

A.1.4 Results

Table A.1 shows the average response times (along with the 95% confidence intervals) in milliseconds for each of the interactions under the following conditions:

1. No background load: This case shows the baseline response time for each interaction.

2. No priority: The background load is present, but neither of the stages enforce priority scheduling based on the metadata.

3. Priority in first stage: The background load is present, and the first stage (the web server) enforces priority scheduling based on the metadata.
4. Priority in both stages: The background load is present, and both stages enforce priority scheduling based on the metadata.

As further illustration of the results, we show in Figure A.1 the response times, sorted in descending order, for the execution of the 100 requests of the search-request interaction under the four cases as described above.

<table>
<thead>
<tr>
<th>Interaction</th>
<th>No background load</th>
<th>No priority</th>
<th>Priority in 1st stage</th>
<th>Priority in all stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin-confirm</td>
<td>60 (±0.2)</td>
<td>1936 (±3.8)</td>
<td>1993 (±38)</td>
<td>342 (±71)</td>
</tr>
<tr>
<td>admin-request</td>
<td>59 (±0.01)</td>
<td>1617 (±120)</td>
<td>868 (±85)</td>
<td>68 (±13)</td>
</tr>
<tr>
<td>best-sellers</td>
<td>918 (±49)</td>
<td>3173 (±986)</td>
<td>3016 (±234)</td>
<td>940 (±33)</td>
</tr>
<tr>
<td>buy-confirm</td>
<td>85 (±1.3)</td>
<td>1951 (±36)</td>
<td>1992 (±67)</td>
<td>1457 (±131)</td>
</tr>
<tr>
<td>buy-request</td>
<td>60 (±1)</td>
<td>1930 (±4.5)</td>
<td>1915 (±59)</td>
<td>81 (±36)</td>
</tr>
<tr>
<td>customer-reg</td>
<td>55 (±1.2)</td>
<td>931 (±88)</td>
<td>61 (±1.5)</td>
<td>60 (±1.6)</td>
</tr>
<tr>
<td>home</td>
<td>61 (±1.7)</td>
<td>1737 (±93)</td>
<td>1095 (±102)</td>
<td>63 (±2.2)</td>
</tr>
<tr>
<td>new-product</td>
<td>81 (±1.7)</td>
<td>1933 (±3)</td>
<td>1969 (±28)</td>
<td>85 (±4)</td>
</tr>
<tr>
<td>order-display</td>
<td>60 (±0.8)</td>
<td>1930 (±3)</td>
<td>1970 (±4)</td>
<td>64 (±4)</td>
</tr>
<tr>
<td>order-inquiry</td>
<td>40 (±0.01)</td>
<td>42 (±2.2)</td>
<td>40 (±1)</td>
<td>40 (±0.3)</td>
</tr>
<tr>
<td>product-detail</td>
<td>60 (±0.6)</td>
<td>1516 (±127)</td>
<td>966 (±100)</td>
<td>68 (±14)</td>
</tr>
<tr>
<td>search-request</td>
<td>60 (±0.03)</td>
<td>1533 (±127)</td>
<td>987 (±102)</td>
<td>61 (±0.7)</td>
</tr>
<tr>
<td>search-result</td>
<td>670 (±0.6)</td>
<td>2628 (±314)</td>
<td>2528 (±5.3)</td>
<td>671 (±1.5)</td>
</tr>
<tr>
<td>shopping-cart</td>
<td>70 (±0.9)</td>
<td>1931 (±4)</td>
<td>1984 (±6)</td>
<td>217 (±40.5)</td>
</tr>
</tbody>
</table>

Table A.1: Average Response Time and 95% Confidence Interval (in milliseconds) for the TPC-W Interactions under High Background Load

Table A.1 and Figure A.1 reflect the behavior under a background load that pushes the system into overload. The same results for a moderate background load are shown in Table A.2 and Figure A.2.
Figure A.1: Response Time Distribution (Sorted in Descending Order) for Search-Request Interaction (High Background Load)
<table>
<thead>
<tr>
<th>Interaction</th>
<th>No background load</th>
<th>No priority</th>
<th>Priority in 1st. stage</th>
<th>Priority in all stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin-confirm</td>
<td>60 (±0.2)</td>
<td>95 (±6)</td>
<td>90 (±6)</td>
<td>65 (±1.3)</td>
</tr>
<tr>
<td>admin-request</td>
<td>60 (±0.2)</td>
<td>92 (±6)</td>
<td>65 (±2.7)</td>
<td>60 (±0.15)</td>
</tr>
<tr>
<td>best-sellers</td>
<td>918 (±49)</td>
<td>1092 (±165)</td>
<td>1137 (±158)</td>
<td>912 (±0.9)</td>
</tr>
<tr>
<td>buy-confirm</td>
<td>85 (±1.3)</td>
<td>136 (±6)</td>
<td>123 (±6)</td>
<td>94 (±1.8)</td>
</tr>
<tr>
<td>buy-request</td>
<td>60 (±1)</td>
<td>103 (±7)</td>
<td>99 (±6)</td>
<td>63 (±1.7)</td>
</tr>
<tr>
<td>customer-reg home</td>
<td>55 (±1.3)</td>
<td>78 (±4.4)</td>
<td>62 (±2.6)</td>
<td>59 (±1.1)</td>
</tr>
<tr>
<td>home</td>
<td>61 (±1.9)</td>
<td>98 (±6.2)</td>
<td>82 (±5.5)</td>
<td>62 (±2)</td>
</tr>
<tr>
<td>new-product</td>
<td>81 (±1.7)</td>
<td>125 (±9.6)</td>
<td>101 (±7)</td>
<td>84 (±3.4)</td>
</tr>
<tr>
<td>order-display</td>
<td>60 (±0.8)</td>
<td>102 (±6.9)</td>
<td>101 (±6.5)</td>
<td>62 (±1.5)</td>
</tr>
<tr>
<td>order-inquiry</td>
<td>40 (±0.01)</td>
<td>40 (±0.15)</td>
<td>40 (±0.01)</td>
<td>40 (±0.01)</td>
</tr>
<tr>
<td>product-detail</td>
<td>60 (±0.6)</td>
<td>94 (±6)</td>
<td>64 (±2.4)</td>
<td>60 (±0.2)</td>
</tr>
<tr>
<td>search-request</td>
<td>60 (±0.04)</td>
<td>97 (±6.3)</td>
<td>65 (±2.8)</td>
<td>60 (±0.14)</td>
</tr>
<tr>
<td>search-result</td>
<td>670 (±0.62)</td>
<td>715 (±19.7)</td>
<td>728 (±11.8)</td>
<td>667 (±3.2)</td>
</tr>
<tr>
<td>shopping-cart</td>
<td>70 (±0.86)</td>
<td>110 (±6.2)</td>
<td>83 (±4.1)</td>
<td>73 (±1.1)</td>
</tr>
</tbody>
</table>

Table A.2: Average Response Time and 95% Confidence Interval (in milliseconds) for the TPC-W Interactions under Moderate Background Load
Figure A.2: Response Time Distribution (Sorted in Descending Order) for Search Request (Moderate Background Load)
A.1.5 Discussion

The results overall confirm the benefits of multi-tier priority enforcement. With priorities enforced at both stages the response times approximate those under no load, and they are substantially better than those in the absence of priorities or in the presence of priorities only at the first stage. The results for single-tier priority enforcement are better than with no priorities, but inferior to using priorities at both stages. The differences are more outspoken in the case of overload, but remain present even under more moderate loads. Given that Causeway allows multi-tier priority propagation without modification of the application and without noticeable overhead, we argue that this serves as a convincing demonstration of its merits.

More detailed inspection of the results on a per-interaction basis leads to some additional observations. In Table A.1 we see that for a large number of the interactions, the response time under load with multi-tier priorities is almost identical to the response time under no load. For a few interactions, however, the response under load is higher, even with the priorities. This observation is explained by the fact that the background load acquires read locks on a certain table in the database, and the fact that the interactions that show a slowdown under load acquire an exclusive lock on that table. As a result, independent of priorities, the foreground interactions need to wait for all current readers to finish before they can proceed at the database. Under overload, there can be a large number of such reads in progress, explaining the marked increases in response time for the admin-confirm, buy-confirm, buy-request and shopping-cart interactions. For the moderate load where only a very few such readers are present, the differences almost vanish (see Table A.2). For foreground interactions that have no conflicts with the background load, there is almost no difference between the the no-load case and the case of load with multi-tier priorities.
Bibliography


call graph execution profiler. In *SIGPLAN Symposium on Compiler Con-

[Hal92] Robert J. Hall. Call path profiling. In *Proceedings of the 14th International

[HG93] Robert J. Hall and Anthony J. Goldberg. Call path profiling of monotonic
program resources in UNIX. In *Proceedings of the USENIX Summer Tech-

[IBB+04] Rebecca Isaacs, Paul Barham, James Bulpin, Richard Mortier, and
Dushyanth Narayanan. Request extraction in Magpie: events, schemas and
temporal joins. In *SIGOPS EW'04: ACM SIGOPS European Workshop*,
September 2004.


[JVM] JVM(TM) Tool Interface 1.0.38. At http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.


