Performance Assertion Checking

by

Sharon Esther Perl

S.M., Massachusetts Institute of Technology (1988)
B.S.E., University of Pennsylvania (1984)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the
Massachusetts Institute of Technology
September 1992

©Massachusetts Institute of Technology 1992. All rights reserved.

Performance Assertion Checking

by

Sharon Esther Perl

Abstract. *Performance assertion checking* is an approach to describing and monitoring the performance of complex software systems. The idea is simple: system implementors write assertions that capture their expectations for performance, the system is instrumented to collect performance data, and then the assertions are checked automatically against the data to detect violations signifying potential performance bugs. Because performance assertions provide a means of filtering data based on expectations, they form a good basis for tools. Data indicating that a system is performing as expected can be discarded automatically, while data indicating potential problems can be brought to the attention of a person.

Performance assertions are useful for performance regression testing, continuous system monitoring, and performance debugging. Also, the act of writing precise performance assertions helps designers and implementors to understand better their own expectations for performance, and the guarantees they can make about their systems.

PSpec is a language and a set of tools that embodies the idea of performance assertion checking. A PSpec user writes performance assertions in the PSpec language. These are input, along with a monitoring log, to a *checker* tool that reports which assertions fail to hold for the log. Another tool called the *solver* helps to fill in constants in assertions using data in a monitoring log. PSpec has been used to find performance bugs in the runtime system of a new parallel programming language, providing evidence of its utility.

Keywords: performance assertion checking, performance testing, performance debugging, performance monitoring, performance specification, regression testing, software performance, PSpec

Thesis Supervisor: Professor William E. Weihl
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

During a fateful conversation about performance debugging tools (and the lack thereof) my advisor, Bill Weihl, asked, "how about a tool that would check performance assertions?" It has taken almost four years to figure out what this question might mean and then to come up with an answer. Many people have helped along the way, and I am pleased to be able to acknowledge their contributions. My deepest appreciation and gratitude go to:

Bill Weihl, who is everything a graduate student could want in an advisor, for his guidance, moral support (and financial support), friendship, and for maintaining his sanity during the past year while he was up for tenure, thereby helping me to maintain mine in my last year of graduate school.

John Guttag, for being an involved and helpful member of my thesis committee, for reminding me of the big picture when I got lost in the details, for having just the right words of wisdom when I needed them, for his eternal optimism and good cheer, and last (but not least) for filling a serious gap in my education concerning the history and virtues of the Larch tree.

Roy Levin, whose participation on my thesis committee was above and beyond the call of duty, for making possible and enjoyable my three summers, one fall, one winter and one spring at DEC's Systems Research Center (greatly stretching the notion of a "summer intern"), for acting as my advisor in residence at SRC, and for sharing his breadth of knowledge and good judgment about systems, from the lowest-level details to the highest-level concepts.

Greg Nelson, for his unwavering and enthusiastic support of this research since I first told him about it over a beer at Rudy's, for many significant technical contributions (including the idea for the solver), for teaching me not to be afraid of simplicity, and for helping me find the motivation both to keep going and to finish.

Adrian Colbrook, for his energetic participation in the Prelude experiments that provided supporting evidence for the practicality of this research, and for much interesting gossip and philosophical discussion.

Eric Brewer, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang, for their help in using Proteus and writing Prelude performance specifications. Particular thanks to Eric for answering many Proteus questions and for helping to find the Prelude performance bugs.

Butler Lampson, for his continued support from the inception of this research, and for conversations at a couple of critical junctures that helped me to move forward when I might have remained stuck.

Bob Taylor, for making possible my extended stays at DEC SRC (not to mention my return).

Tim Mann, Garret Swart and Andy Hisgen, members of SRC's Echo project, for their interest and cooperation during the early phases of the research. Though the results of my Echo experiments are not an explicit part of this dissertation, many of the important ideas underlying the research were developed during that time.

Bruce Leban, Dorothy Curtis, and Hal Murray, for promptly proofreading thesis drafts. Each had helpful suggestions for improving the presentation and each managed to find typos that nobody else found. Thanks also to Bruce for some handy Latex hacking.

Sue, Jack, Eleanor and Jonathan Owicki, for making me feel like one of the family while I lived with them in Palo Alto, and to Sue also for many interesting technical discussions.

Marilyn Pierce, whose cheerful and supremely competent handling of all the administrative details associated with being a student makes every EECS graduate student's life easier.

My parents: to my father, for constantly asking (in his inimitable way) why I didn't just write the thesis and be done with it, and to my mother, for understanding the answer. Their abiding love and support throughout my life have instilled the deeply-rooted self-esteem that allowed me to continue through the difficult times when self-doubts might have stopped me. A child cannot receive a more valuable gift from her parents.

This research was supported by the NSF under grants CCR-8716884 and CCR-8822158, by DARPA under contract N00014-89-J-1988, and by Digital Equipment Corporation.

Contents

1 Introduction 13
- 1.1 The Idea . . . 14
- 1.2 The Approach . . . 14
- 1.3 Relation to Other Work . . . 16
  - 1.3.1 Assertion Checking . . . 16
  - 1.3.2 Supporting Performance Work . . . 17
  - 1.3.3 Complementary Performance Work . . . 18
- 1.4 Road Map . . . 19

2 Performance Assertion Model 21
- 2.1 Concepts . . . 21
- 2.2 Expressing Performance Assertions . . . 23
  - 2.2.1 Example: A Simple File System . . . 23
  - 2.2.2 Response Time . . . 24
  - 2.2.3 Throughput . . . 27
  - 2.2.4 Utilization . . . 28
  - 2.2.5 Workload Properties . . . 29
- 2.3 Evaluation of the Model . . . 33

3 The PSpec Language 35
- 3.1 Why a Special-Purpose Language? . . . 35
- 3.2 Overview . . . 36
- 3.3 Language Description . . . 37
  - 3.3.1 Simple Declarations and Assertions . . . 37
  - 3.3.2 More on Identifying Intervals . . . 39
  - 3.3.3 More on Expressing Assertions . . . 41
  - 3.3.4 More on Defining Metrics . . . 42
  - 3.3.5 Specifications . . . 46

3.3.6 Measurement Error 47
3.3.7 Summary of Features 48
3.4 Language Design Choices and Tradeoffs 49
3.4.1 Intervals 50
3.4.2 Aggregate Expressions 52
3.4.3 Metric Definitions 53
3.4.4 Non-extensibility 53

4 Tools 55
4.1 Checker 55
4.2 Solver 57
4.2.1 Overview 57
4.2.2 Easy Example 58
4.2.3 Harder Example 59
4.2.4 Using the Solver Intelligently 61
4.3 Implementation 62
4.3.1 Parsing and Type Checking 63
4.3.2 Evaluation 63
4.4 Other Tools: Wish List 69

5 Experience 71
5.1 Testbed 71
5.2 Prelude Performance Specifications 73
5.2.1 Response Time Specifications 73
5.2.2 Workload Specifications 75
5.2.3 Scheduler Specification 76
5.3 Prelude Performance Bugs 78
5.4 Evaluation 79
5.5 Methodology 80
5.5.1 Process 80
5.5.2 Hints for Writing Performance Specifications 82

6 Extensions and Future Directions 85
6.1 "Real" Systems 85
6.2 Language Extensions 87
6.2.1 Computing State From a Log 87
6.2.2 Virtual Events 90
6.3 Performance Specifications as Interface Documentation 91

7 Conclusion 95
7.1 Summary of Contributions 95
7.2 Directions for Future Research 97

References 99

A PSpec Language Reference Manual 105
A.1 Introduction and Definitions 105
A.2 Types 106
A.3 Declarations 106
A.3.1 Constants 106
A.3.2 Event Types 107
A.3.3 Interval Types 107
A.4 Assertions 108
A.5 Solve Declarations 108
A.6 Specifications 109
A.7 Expressions 109
A.7.1 Literals 110
A.7.2 Triple Constructors 110
A.7.3 Mappings 110
A.7.4 Field Access 111
A.7.5 Time Units 111
A.7.6 Arithmetic Operations 111
A.7.7 Relational Operations 112
A.7.8 Logical Operations 114
A.7.9 Operations on Intervals 114
A.7.10 Operations on Events 114
A.7.11 Aggregates 114
A.8 Grammar 117

List of Figures

1.1 The PSpec approach 15

2.1 Sample monitoring log 22
2.2 File system interface 24
2.3 Events logged for the file system 25

3.1 Mapping operations 46
3.2 Example of combining mappings 46

4.1 Estimating *PerByteTime* and *Overhead* by line-fitting 60
4.2 Algorithm for generating interval log stream 65
4.3 Algorithm for evaluating specifications 68

5.1 A response time specification for disabling of interrupts . . . 74
5.2 A workload specification: Interprocessor sends 75
5.3 Scheduler specification 77

A.1 Operator precedence 110
A.2 Arithmetic operations on triples 113
A.3 Boundary case results for aggregate operators 116

Chapter 1

Introduction

Is it not strange that desire should so many years
outlive performance?
—William Shakespeare, Henry IV, Part II

Systems often do not perform as well as we would like or expect. Designing for good performance ahead of time and then debugging and testing performance once a system is implemented are difficult tasks. One problem is that systems often develop performance bugs that go unidentified for long periods of time. By *performance* I mean any measure of resource usage, for example, elapsed time, throughput or utilization. Designers and implementors have—or should have—expectations for how their systems will perform. A *performance bug* is a failure of the system to meet those expectations. Performance bugs may develop because of changes in the software (perhaps enhancements or bug fixes), changes in the hardware platform, or changes in the workload presented by users. Workload changes in particular may invalidate assumptions that the implementation relies upon to achieve good performance.

Performance bugs go unidentified for several reasons. One is that the users of a system may not know what performance to expect, and so they cannot tell that performance is worse than it should be. Another reason is that the gross effects of a performance bug may initially be small and so the presence of the bug is not immediately obvious. A third reason is that, while everyone may be aware that something is wrong with a system's performance, the cause may be low-level and difficult to track down.

1.1 The Idea

The main idea behind this research is that performance assertions can help to detect performance bugs early on. The idea is simple: system implementors write assertions that capture their expectations for performance, the system is instrumented to collect performance data, and then the assertions are checked automatically against the data to detect violations signifying potential performance bugs.

Performance assertions are useful for:

- Performance regression testing: once the performance of a program is understood, it can be captured with a set of performance assertions. When the program is changed, the assertions can be rechecked to ensure that the performance still meets expectations.
- Continuous system monitoring: performance assertions can be checked routinely during normal system use to determine whether performance is meeting expectations and whether workloads satisfy the assumptions that were made during system design.
- Performance debugging: successively more detailed performance assertions may be helpful for pinpointing the location of performance problems in the system.
- Clarifying expectations: the act of writing precise performance assertions helps designers and implementors to understand better their own expectations for performance, and what they can or cannot guarantee about their systems.

Performance testing and debugging often involve processing vast quantities of data. One reason why these activities receive inadequate attention is the dearth of good tools for dealing with the voluminous data. Performance assertions provide a means of filtering data based on expectations, and so provide a good basis for tools. Data indicating that a system is performing as expected can be discarded automatically, while data indicating potential problems can be recognized and brought to the attention of a person.

1.2 The Approach

PSpec is a language and a set of tools that embodies the idea of performance assertion checking. It is intended to be useful in concurrent systems, ranging

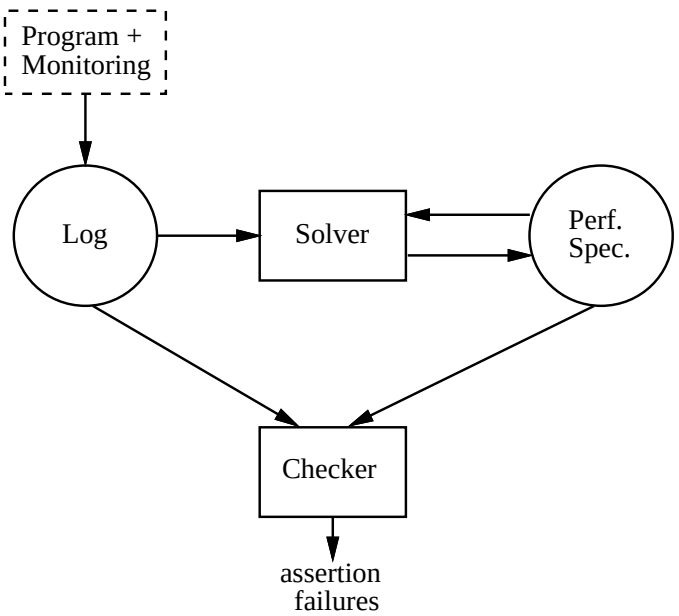

Figure 1.1: The PSpec approach

from multitasking uniprocessor systems to small-to-medium scale multiprocessors to distributed systems. It may be applicable to highly parallel systems as well, but no particular consideration was given to the requirements of such systems during the design.

The PSpec approach is illustrated in Figure 1.1. The components of the approach are:

- Monitoring logs: A monitoring log is an abstraction of a program's execution that contains everything that is relevant about its performance for the purpose of assertion checking. The user supplies an augmented version of a program that generates a monitoring log for each run. The mechanism for generating monitoring logs is not part of PSpec. Instead, PSpec defines a simple, general log interface that can be implemented on top of many available monitoring systems.

- Performance specifications: These are documents containing performance assertions, supplied by the user and written in the PSpec language. The language is a notation for expressing predicates about

monitoring logs. The language can express assertions about many common kinds of performance metrics such as elapsed time, throughput, utilization, and workload characteristics, given the appropriate kinds of events in logs.

- The checker: Provided with a performance specification and a monitoring log, the checker reports which assertions fail to hold for the run represented by the log.
- The solver: The solver helps a specification writer to fill in constants in assertions using data in a monitoring log. The idea is that specifications will sometimes contain numeric constants whose values must be determined in part by measuring the system being specified. The solver can take a specification with "unknown" symbolic constants and, in particular situations, estimate values for those constants using the data in a log. In essence, the solver packages up simple equation solving capabilities and a linear regression solver in a form that is convenient to use with specifications and monitoring logs.

The initial versions of the PSpec tools were implemented and tested in the context of a multiprocessor simulator that provided a convenient testbed for the research. Future plans call for applying the PSpec approach to real (non-simulated) systems.

1.3 Relation to Other Work

The literature on computer systems performance is vast and varied. To put PSpec in context, there are two classes of related work on performance that I will discuss briefly. The first class, supporting performance work, provides the necessary foundation for PSpec. The second class, complementary performance work, either has different goals than the PSpec work, or takes a different and complementary approach to the same problem. Before talking about the relation to other performance work, it is worth mentioning the connection to work on functional assertion checking, which provided the initial inspiration for the PSpec work.

1.3.1 Assertion Checking

Performance assertions are somewhat analogous to functional assertions. Several programming languages (including Euclid [22] and ANSI C [19]) provide constructs that allow users to write assertions that are checked during

program execution. Such assertion checks are generally used to test simple invariants about the state of a computation. To write a functional assertion, typically, a programmer annotates code with an *assert statement* that tests a predicate about the state of the computation and terminates the program if the predicate evaluates to false. Although terminating the program might seem rather drastic, the failure of a functional assertion generally indicates a serious bug that would invalidate the remainder of the computation.

Performance assertions are like functional assertions in that they express predicates about a program that can be checked for an execution. They differ from functional assertions in two ways. First, performance assertions may apply to a series of program states rather than to a single state (for example, to express something about the elapsed time for a computation). Second, we probably do not want their failure to result in a crash; the computation can complete correctly—it just might be slower or consume more resources than expected.[1]

1.3.2 Supporting Performance Work

Supporting performance work falls into two categories: monitoring, and performance characterization.

Monitoring, as I use the term, encompasses data observation (event recognition) and collection (log production) facilities.[2] As mentioned above, monitoring is a necessary but separable component of PSpec. I have assumed that a monitoring facility would be available, or could be built, to produce event logs against which performance specifications can be checked. Monitoring is useful for much more than performance assertion checking—for example, it is used for tracing, timing, tuning, coverage analysis, and performance characterization. Consequently, there is much active research on mechanisms for efficient monitoring in all kinds of systems. The PSpec notion of a log is general enough to fit many existing monitoring systems with little work. For examples of recent work on monitoring, see [3, 12, 20, 24, 28, 32, 33, 38, 43]. General discussions of monitor design may be found in [11] and [18], and in other textbooks on performance measurement.

Performance characterization includes work on defining performance metrics, on characterizing workloads, and on techniques for determining values

[1] In a system with hard or soft real-time constraints a performance assertion failure may indeed have consequences as serious as a functional assertion failure. This research has not focused on such systems.

[2] Some authors define monitoring to include data analysis and presentation.

of metrics. Work in this area can provide guidance to specification writers in developing and expressing expectations about performance. Work on linear regression analysis has formed the basis for the design of the solver and can help specification writers to understand how to set up experiments to obtain useful results from the solver. Many textbooks on performance include discussions of performance characterization or linear regression, for example, [10, 11, 18, 23].

1.3.3 Complementary Performance Work

PSpec is an approach to performance testing and debugging based on automatically checking expectations captured in performance assertions. Other work in performance testing and debugging has taken different approaches. Profiling tools, such as gprof [14], pixie [9], or Quartz [2], provide breakdowns of where time is spent in programs. Such tools are useful for performance debugging because the information they provide can be examined by someone who has an idea of where the time should go to determine whether things are as expected. In some sense, using profiler output is like doing manual performance assertion checking. Visualization tools provide pictorial displays of performance data—either static (e.g., graphs) or dynamic (e.g., animations) [5, 13, 16, 25, 27]. They are useful because pictures can clearly convey effects that may be hidden in a mass of numbers. Visualization tools can help the user to discover the existence of bugs where he or she did not expect them. As with profiling tools, a person must examine the output to find problems—the process is not automated. For this reason such tools cannot help with continuous monitoring.

Performance tuning involves finding and eliminating bottlenecks in order to improve performance. The profiling tools mentioned above are also useful for tuning in addition to debugging. PSpec does not offer any help with tuning. In the tuning process the user does not necessarily have any expectations about performance. In fact, tuning a program based on expectations of where the bottlenecks are is generally a bad idea—programmers' predictions are often wrong (e.g., see [31]). The tuning process would most likely precede the writing of performance assertions.

PSpec also does not offer any help with performance prediction. Performance prediction usually involves either analysis or simulation—often before a system has been implemented—and, again, is aimed at developing expectations about performance rather than checking them. Some work in this area is reported in [1, 8, 36, 37]. Once performance predictions have been

made they can be captured in performance assertions so that they can be checked against actual performance.

Finally, work on verifying timing properties of programs has similar high-level goals to PSpec but is a completely different approach with different capabilities. It is similar in the sense that the goal is to discover whether a system performs as expected or required. However, verification work is aimed at proving that a program or system meets time constraints, usually by analyzing its source code. Generally, the properties that are proved are bounds, and the technique is used when meeting performance bounds is critical rather than just preferable. Considerations of real-time systems have motivated much of this work. Because predictable performance is so crucial, such systems often rule out implementation techniques (such as virtual memory) common in non-real-time systems. Building the systems for predictable performance makes it easier to verify bounds; many non-real-time systems do not have provable time bounds. Proving performance properties that are not bounds (for example, distributions) is much more difficult. Even for proving bounds in real-time systems, verification is generally difficult and expensive. Though it would be nice to be able to prove performance properties of programs—thus avoiding the overhead of checking during operation—it is not a practical approach at this time for non-real-time systems of any complexity. Some work on proving performance properties includes [4, 15, 26, 35, 34, 40].

1.4 Road Map

The remainder of this dissertation is divided into six chapters and one appendix. Chapter 2 presents the model underlying the PSpec language, showing how it allows a variety of performance properties to be captured, and also discussing its shortcomings. Chapter 3 presents the PSpec language through examples, and discusses how the choices and tradeoffs made in designing the language affect design goals such as expressiveness, readability, and efficiency. Chapter 4 describes the checker and solver tools that have been designed and implemented, and considers some other tools that might be useful. The discussion includes a description of the tools' functionality, as well as interesting aspects of their implementation. Chapter 5 describes my experience using PSpec to find performance bugs in the runtime system of a new parallel programming language. Chapter 6 describes possible short-term extensions to the work to explore further the usefulness of the PSpec approach, and to correct some of the remaining problems with the language.

The chapter also includes a discussion of a direction for longer-term future research. Chapter 7 concludes by summarizing the contributions of the work and mentioning additional avenues for future research. The appendix contains the reference manual for the language.

Chapter 2

Performance Assertion Model

In this chapter we examine the model underlying the PSpec language. The model is basically an abstraction built on top of monitoring logs that helps us to capture performance properties of interest for writing performance assertions. These properties include elapsed times for computations (e.g., the elapsed time for a read operation in a file system), throughputs (e.g., the number of bytes per second that can be read from files), measures of utilization (e.g., the percentage of time that a processor is idle), and workload characteristics (e.g., the frequency of read requests). After describing the model, I will discuss each of these kinds of properties in detail and give examples of how they are expressed using the model and informal mathematical expressions. The chapter concludes with an evaluation of the model, discussing the one shortcoming that has become apparent from working with examples.

The discussion in this chapter is entirely in terms of the model, which is more abstract and more general than the PSpec language. The language imposes certain restrictions on the model, both to enable assertions to be checked efficiently and to keep the language simple. The next chapter describes the PSpec language and discusses the motivation for the restrictions and their effect.

2.1 Concepts

To begin, we need a way to talk about the execution of a program whose performance is of interest. As mentioned in the previous chapter, a *monitoring log* is an abstraction of a program's execution that captures the information that is relevant for expressing performance assertions.

StartRead (tid = 102, ts = 1)
InterruptsOff (pid = 1, tid = 105, ts = 2)
InterruptsOn (pid = 1, ts = 3)
InterruptsOff (pid = 2, tid = 120, ts = 4)
CacheHit (tid = 102)
StartRead (tid = 104, ts = 5)
EndRead (tid = 102, ts = 6)
InterruptsOn (pid = 2, ts = 7)
EndRead (tid = 104, ts = 8)
StartRead (tid = 104, ts = 9)
CacheHit (tid = 104)
EndRead (tid = 104, ts = 10)

Figure 2.1: Sample monitoring log

A log is a sequence of *typed events*. An event type is a list of named *attributes*. Event types are unique (branded by a name). An event supplies a value for each attribute of its type. For example, we could have an event type *InterruptsOff* with the attributes *pid*, *tid*, and *ts*. Events of this type correspond to the disabling of interrupts for a processor, with values of the attributes identifying the processor, the thread that caused interrupts to be disabled, and the time at which the event occurred. Similarly, we could have another event type *InterruptsOn* with *pid* and *ts* attributes, corresponding to reenabling interrupts on the processor identified by *pid* at time *ts*.

Figure 2.1 shows a representation of a sample log, containing events of types *InterruptsOff* and *InterruptsOn*, as well as some other events with suggestive type names. *StartRead* and *EndRead* events correspond to the initiation and completion of read requests to a file system. *CacheHit* events appear between *StartRead* and *EndRead* events for a read request whenever the request can be satisfied from the file system cache. In this example, all of the event types except *CacheHit* have timestamps. All events are ordered in the log, whether or not they have timestamps.

Events record useful information by themselves, but as suggested by the interrupt and read cases, sometimes we are interested in subsequences of the log between two events. For example, we might be interested in the time that elapses between an *InterruptsOff* event and the corresponding *InterruptsOn* event, or we might be interested in whether a *CacheHit* event

appears between a *StartRead* for a request and its corresponding *EndRead*. For this reason we introduce the notion of an *interval*.

An interval consists of all events in a log between a designated *start event* and an *end event*. Like events, all intervals are of some named interval type. Just as an event type has named attributes, an interval type has named *metrics* that record values of interest for intervals of the type. Metrics are computed from the events that comprise an interval. For example, we could have an *InterruptsDisabled* interval type for intervals that start at some *InterruptsOff* event and continue through the corresponding *InterruptsOn* event, with a metric called *time* whose value is the difference of the timestamps of the start and end events for the interval.

An interval is ordered with respect to other intervals and events by its end event.[1] Two intervals with the same end events, or an interval and its own end event, are not ordered with respect to each other. Thus, given a set of interval type definitions, a log can be viewed as a partially ordered set of intervals and events.

Intervals are the primary abstraction used in writing performance assertions. The typical *modus operandi* for writing assertions is to figure out what is to be asserted about the performance of a program, define interval types that capture the metrics needed for the assertions, and then write predicates that apply over the set of intervals of the defined types.

2.2 Expressing Performance Assertions

Why should you believe that this model with logs, events, and intervals is a good basis for writing performance assertions? In this section I will attempt to persuade you that it is by showing how it allows a variety of different kinds of performance properties to be expressed.

2.2.1 Example: A Simple File System

In the following discussion I will primarily use a single setting for the examples to illustrate performance metrics. The setting is a UNIX-style file system, simplified for our current purpose. The file system provides operations to create, open, and close files, to read and write sequences of bytes, to reposition the file pointer (seek) for the next read or write operation, and to truncate files to a given length. The interface is shown in Figure 2.2.

[1] This definition permits efficient recognition of intervals in a log, as will become apparent in Chapter 4.

Create (fname): fid
Create a new file with name *fname*. Returns file identifier *fid*.
Open (fname): fid
Open existing file with name *fname*. Returns file identifier *fid*.
Read (fid, buf, size)
Read the next *size* bytes from file *fid*, placing the data in *buf*.
Write (fid, buf, size)
Write the *size* bytes from *buf* to the current position in file *fid*.
Close (fid)
Close file *fid*.
Seek (fid, pos)
Change the current position in file *fid* to position *pos*.
Truncate (fid, len)
Truncate file *fid* to *len* bytes.

Figure 2.2: File system interface

In terms of implementation, files are stored on a disk and cached in memory to improve the performance of Read and Write operations. Writes put the data in the cache and a background thread flushes new data to disk periodically. Reads check the cache before going to disk for data.

In discussing the performance metrics for the file system, we take the point of view of one of its implementors. Thus we have knowledge of the inner workings and the ability to instrument whatever parts of it we like in order to collect event logs for performance assertion checking. Some of the metrics we define will be "interface level" (such as elapsed times for operations) while others will be "implementation level" (such as the cache hit rate).

The events that we log include calls and returns for each of the file system operations and hits in the cache during Read operations (they are listed, with their attributes, in Figure 2.3). Other events will be introduced as needed.

2.2.2 Response Time

The *elapsed time* for a computation is the amount of real time between the initiation of the computation and its completion. As we saw above, we

StartCreate (tid, ts), EndCreate (tid, ts),
StartOpen (tid, ts), EndOpen (tid, ts),
StartRead (tid, size, ts), EndRead (tid, ts),
StartWrite (tid, size, ts), EndWrite (tid, ts),
StartClose (tid, ts), EndClose (tid, ts),
StartSeek (tid, pos, ts), EndSeek (tid, ts),
StartTrunc (tid, len, ts), EndTrunc (tid, ts)
CacheHit (tid)

Figure 2.3: Events logged for the file system

can express an elapsed time metric in the PSpec model by introducing an interval type with a *time* metric. For example, we can compute the elapsed time for file system Read operations by defining such an interval type, *Read*, demarcated by *StartRead* and *EndRead* events for the same thread. Then we can express assertions on the bounds for elapsed times of *Read* intervals or statistical assertions about the elapsed times. A bounds assertion might be:

> *For all Read intervals r, r.time is at most one second.*[2]

Some examples of statistical assertions are:

> *Over all Read intervals r, the average value of r.time is at most twenty milliseconds,*

and

> *The fraction of Read intervals in a log whose elapsed time is more than ten milliseconds is at most 0.5.*

We can only expect these statistical assertions to hold for logs that have a sufficient number of Read intervals. To avoid spurious reports of assertion failures we could ensure that we only check the assertions against logs that contain enough data. Alternatively, we could modify the assertions to express the condition. If we determine that the statistical results should hold when there are at least thirty Read intervals, we can rewrite the assertion about averages above to say:

[2] I use the dot notation, as in *r.time*, to refer to a metric of an interval or an attribute of an event.

> *If there are at least thirty Read intervals, then over all Read intervals r, the average value of r.time is at most twenty milliseconds.*

Sometimes instead of writing elapsed time assertions for entire operations, we would like to write assertions about the elapsed time for subcomputations in an operation. For example, suppose that the implementation of a Read operation involves obtaining a lock internal to the file system. We might like to assert that the wait times for the lock during *Reads* are short in most cases (say, that the wait is at most one millisecond for ninety percent of the operations). We can do this by adding another metric to the *Read* interval type, *lockwait*, that records the amount of time spent waiting for the lock during an interval.[3] Using the *lockwait* metric we can write the assertion:

> *The fraction of Read intervals whose lockwait metric is less than one millisecond is at least 0.9.*

Another kind of elapsed time assertion gives a model of the elapsed time for an operation in terms of parameters of the operation, constants, and times for subcomputations that comprise the operation. For example, if we believe that the elapsed time for a Read that hits entirely in the cache is a particular function of the size of the Read and the time spent waiting for the lock, then it would be useful to write and check such an assertion to discover whether our mental model is correct.

First, we add another metric to the *Read* interval type, *hit*, that indicates whether a Read operation hit in the cache. This metric will be true for intervals that contain a *CacheHit* event. We also add a metric *size* to record the size of the Read; this value can be obtained from the *StartRead* event. Then we can write an assertion about the model for the elapsed time. Assuming that *PerByteTime* and *Overhead* are constants whose values are known[4], we can assert:

> *For all Read intervals r where r.hit is true, r.time is at most*
>
> $$PerByteTime \times r.size + Overhead + r.lockwait.$$

[3] To do this we introduce event types corresponding to the requesting and granting of the lock, and define an interval type, *LockWait*, starting and ending at these event types. Then we can compute the value of *lockwait* for a *Read* interval by adding up the elapsed times of all *LockWait* subintervals within the *Read* interval.

[4] These constants can be determined in a number of different ways; we will discuss this when we talk about the solver in Chapter 4.

2.2.3 Throughput

Throughput is a measure of the number of units of work that can be accomplished in a given time interval. For example, we might talk about the file system throughput in terms of the number of bytes of data that can be read or written per second. In a transaction processing system throughput might be the number of transactions completed per second. In a network, throughput might be the number of bytes transferred per second.

Suppose we would like to write an assertion for the file system about the average Read throughput (in bytes per second) per thread that reads data over the lifetime of a log.[5] This is equal to the total number of bytes read over the lifetime of the log, divided by the time spanned by the log, and divided again by the number of different threads that read data. Such an assertion could be used for regression testing purposes; the test program would be carefully constructed to set up an appropriate number of threads reading data from specially constructed files.

To write the assertion we must compute each of the three quantities. The total number of bytes read over the lifetime of the log can be obtained by summing, over all *Read* intervals r, the value of $r.size$. The time spanned by the log can be computed by defining an interval type with a *time* metric that starts with the first event in the log and ends with the last. Finally, the number of different threads that read data can be computed by adding a *tid* metric to *Read* intervals (obtaining the value from the *StartRead* event) and creating a set whose elements are the values of the *tid* metrics in all *Read* intervals in a log. The size of this set is the number of different threads that read data.

Sometimes in talking about throughput we would like to take averages over intervals of time smaller than the time spanned by the entire log. For example, we might like to talk about average throughput per active thread over ten-minute intervals or ten-second intervals. A thread is considered active in an interval if it initiates any file system activity in that interval. We might expect a ten-minute average to reflect the steady state throughput per user, while a ten-second average would reflect bursts of activity, if indeed the usage is bursty.

Computing averages over fixed-size intervals (e.g., ten minutes or ten seconds) is straightforward. For example, let us consider how we could express an assertion about the average Read throughput per active thread

[5] The examples in this section are loosely based on a characterization of throughput properties for a real file system [30].

over ten-minute intervals. First, we define an interval type, *TenMinutes*, corresponding to ten-minute intervals. These intervals can be identified either by using the timestamps on events, or by introducing explicit events that are generated in the log every ten minutes. Then for each such ten-minute interval we can compute the average number of bytes read per active thread by adding up the sizes for all Reads requested in the interval and dividing this by the number of active threads in the interval. (We can compute the number of active threads as before—by creating a set of thread identifiers and computing its size.) Finally we can compute the average over all ten-minute intervals of the average number of bytes read per thread in each interval, obtaining the desired metric for use in an assertion.

2.2.4 Utilization

Utilization is a measure of how much capacity of a resource is actually used at a given time or over a given time interval. For example, a measure of utilization for a processor might be the percentage of time in a second that the processor is not idle. A measure of utilization for a network might be the percentage of available bandwidth that is in use over a given time interval.

Suppose that we would like to express an assertion about the utilization of the disk in our filesystem—perhaps that the disk never gets too full, since performance could degrade significantly in that case. In order to write the assertion we need information about disk utilization in the log. One way to get this is to generate events that report the disk utilization every time it changes (any time new disk blocks are allocated or freed). If this is too expensive in terms of logging cost we could instead generate events periodically that report the amount of free disk space. Periodic reports would probably give a close enough approximation to be useful. Assuming that we have events of type *DiskSpace* with attribute *freekbytes* recording the free disk space, it is a simple matter to write an assertion that says that the available disk space is always above some minimum (say, ten percent of the quantity *DiskCapacity*, which can also be recorded in an event). The assertion is:

$$\textit{For all DiskSpace events } d,\ d.\textit{freekbytes} \geq 0.1 \times \textit{DiskCapacity}.$$

Another interesting utilization metric for the file system might be the average processor utilization caused by file system activity. In order to determine this we need information in the log about the processor time expended on file system requests, and also about processor time expended

for background file system activity (e.g., the background thread that flushes the cache to disk). Timestamps in events typically give us information about real time, not processor time, so processor time events must be generated explicitly.

Each time a thread initiates or completes a file system request we could generate an event to record the cumulative amount of processor time used by the thread at that point; call these event types *StartFSCPU* and *EndFSCPU*. Also, for all background threads in the file system, we could generate events of these types each time one of those threads starts or stops computing. Then we could define an interval corresponding to processor time used on behalf of the file system, demarcated by *StartFSCPU* and *EndFSCPU* events for the same thread, and with a metric, *cputime*, whose value is the difference of the cumulative processor times at the start and end of an interval. Adding up the *cputime* metrics from all such intervals gives us the processor time taken by all file system operations in a log, and dividing by the elapsed time spanned by the log then gives us the average processor utilization caused by the file system.

2.2.5 Workload Properties

Implementors often make assumptions about the way their systems will be used in order to optimize their code for the common case. If such assumptions are incorrect, or become incorrect over time as the usage patterns of the system change, performance will degrade. Often such assumptions are not clearly documented, and the implementors themselves may forget their assumptions as time passes. As a result, when the system fails to perform well it is not obvious what the cause is or how to fix it. Also, because of the failure to document these assumptions, clients may use the system in non-optimal ways. If clients had documentation about how to use the system to achieve good performance, or at least received a warning when their usage patterns failed to match the implementors' assumptions, causes of poor performance might be found and fixed earlier than they would be otherwise.

Assertions about workload properties fall into two categories: those that are easy to express purely in terms of the requests at the interface to the system, and those that are not. Often the latter case is due not to limitations of the PSpec model, but rather to the difficulty of describing precisely in any model how to recognize workloads that lead to good (or bad) performance. This is particularly true when the history of requests to a system, by affecting

the system's state, also affects the performance of future requests. This happens, for example, in file systems that have caches or that do read-ahead.[6] One of the strengths of the PSpec assertion checking approach is that we can write either interface-level or implementation-level assertions, depending upon which is more convenient for the properties we want to express.

An example of a workload property that can be expressed in interface-level terms is the rate of Read requests to the file system, either overall or on a per-thread basis. To write the assertion that the average overall Read request rate is at most ten requests per second (or equivalently, that the average time between requests is at least 0.1 seconds), we can define an interval type, *ReadRequestInt*, corresponding to the intervals between Read requests (starting and ending with *StartRead* events). These intervals will have a *time* metric as usual. The assertion is then:

> *Over all ReadRequestInt intervals r, the average value of r.time is at least 0.1 seconds.*

If we want to talk about the average Read request rate per thread, we can define *ReadRequestInt* intervals for each thread, matching start and end events using *tid* attributes. Then we can compute, for each different thread identifier appearing in a Read request interval, the average elapsed time for Read request intervals for that thread, and write the assertion to apply over all such thread identifiers.

An example of a property that is not easy to express in interface-level terms is that files are accessed in such a way as to make the cache an effective implementation strategy. In a sense, an assertion about the expected cache hit rate expresses a property of the workload; for example, if clients only read files that they have not read before, or if they read too many new files before reading old ones, then the hit rate will be poor. This workload property is expressed in terms of an implementation-level concept, the cache.

To write an assertion about the cache hit rate for file system Reads, we need to measure the percentage of Read operations that can avoid going to disk because the required data is in the file system cache. Alternatively, we could define the Read cache hit rate to be the percentage of bytes that are read from the cache rather than from disk instead of the percentage of operations. Either of these metrics is easily expressed in the model. We will use the first definition for illustration.

[6] Read-ahead involves initiating Reads for data that the client might be expected to request in the future based on past requests.

In our file system setting, a *CacheHit* event is generated during a Read operation whenever the Read can be satisfied entirely out of the cache, without going to disk. Earlier, in our discussion of models for elapsed times of Read operations that hit in the cache, we used this fact to define the *Read* interval type with a boolean metric *hit* that is true for Reads that hit in the cache. We can use this interval definition for writing an assertion about cache hit rates. For example, we can assert that:

The fraction of Read intervals whose hit metric is true is at least 0.75.

Another example of a workload property is the amount of concurrency present in requests. For example, the elapsed time for a file system operation that involves obtaining a lock is affected by the presence of other concurrent operations also vying for the lock. We might want to write an assertion about the elapsed time for such an operation under the condition that there are no concurrent requests. Another assertion might say that the level of concurrency is usually low (and thus, even when operations include lock waiting times they are not very long). We can imagine writing such assertions in interface-level terms (identifying operations that overlap in the log) or in implementation-level terms (the amount of contention for the lock). In the PSpec model, these interface-level assertions are difficult to express, while the implementation-level assertions are straightforward. We will look at examples to see why this is true.

An interface-level assertion for file system Reads might be something like: "the elapsed time for a Read operation that is not concurrent with any Write operations is at most $PerByteTime \times size + Overhead$." The question then is, how can we identify Read operations that are not concurrent with Write operations? Three approaches that initially come to mind.

One approach is to write the interface-level assertion with a double quantification over all *Read* and *Write* intervals. Assuming that we have defined a *Write* interval type similar to the *Read* interval type we have been using, we can add metrics to *Read* and *Write* intervals that record the timestamps of their start and end events, say *startts* and *endts*. Then we can write:

For all Read intervals r,
  if for all Write intervals w, $w.endts < r.startts$ *or*
  $w.startts > r.endts$ *then*
    $r.time \leq PerByteTime \times r.size + Overhead.$

This assertion fits with the model, but as we will see in the next chapter, it is ruled out of the PSpec language for efficiency reasons.

A second approach is to introduce a boolean metric for *Read* intervals that will be true whenever there are no overlapping *Write* intervals, and then to write the assertion to apply to all *Read* intervals for which this metric is true. This idea will not work, however, because there is no way to compute the value of this metric in the PSpec model given just the start and end events for Reads and Writes. The problem is that the value for a metric of an interval must be computed using only the events contained inside the interval. So, we can write an expression to compute whether any *Write* intervals start or end inside a *Read* interval, but we cannot compute from a *Read* interval whether there are *Write* intervals that start before the *Read* and end after it. Possible changes to the PSpec model to enable the metric to be computed are discussed in Chapter 6.

A third approach is to generate an event in the log for each *Read* interval indicating whether there are any *Writes* concurrent with the *Read*; this requires keeping track of concurrency in the file system itself. This would enable us to write the assertion, but it is not a very satisfactory solution because of its cost in terms of additional time and complexity in monitoring.

On the other hand, we can express implementation-level assertions, such as "the elapsed time for a Read operation where there is no contention for the file system lock is ... ." This would involve generating an event in the log corresponding to each lock request, indicating whether the lock was immediately available. Such an event is easy to generate, and does not involve introducing new state into the file system as would be required to keep track of concurrency. If we wanted to avoid generating this extra event, we could simply examine the elapsed time for the lock acquisition (using *LockWait* intervals as before) and check whether it was bounded by a small constant. This would accomplish the same purpose, which is to give a model for the elapsed time of a Read in terms of the size of the Read for the case when there is no chance of large variability in the elapsed time due to lock contention.

Some other examples of potentially interesting assertions about the workload for the file system include:

- Many files have short lifetimes.
- Many files accessed are small.
- Many files accessed are read in their entirety.
- Many files accessed are read sequentially.

- Many file opens are for files in the current directory.

Of course, "many," "small," and other such qualitative terms must be made quantitative in order to write the actual assertions. Given that, all of the above assertions can be expressed in PSpec (perhaps by adding some additional events over those already defined).

2.3 Evaluation of the Model

One major shortcoming of the PSpec model has emerged from the examples that I have studied. We just saw an instance of it when we tried to express interface-level assertions about concurrency of Reads and Writes in the file system. The troublesome aspect of the model is that metric values can be computed only from the events or subintervals within an interval, but there are cases where we would like to compute a metric's value using events outside the interval. In particular, it is easy to construct examples where we want to use all of the events from the beginning of the log up to some point in an interval to compute one of its metrics. The reason for this is that we want to compute some aspect of the *state* of the program execution at a given point based on the information in the log. In the example above, the state of interest is whether there is a Write operation pending when a Read starts (or vice versa). In Chapter 5 we will see another example where the ability to compute state is desirable.

Lacking the ability to compute state, the alternative is to record an event containing the state information at the point in the log where it is needed. This solution works, but it is inelegant and potentially expensive, because we have to generate additional events in the log and, perhaps, keep additional state around in the program being monitored. The current notion of an interval as a self-contained unit is appealingly simple, but the model should probably be changed to allow more flexibility in using information that is, after all, already present in a log. Chapter 6 describes how we might fix the model and the effect that this would have on the PSpec language.

Aside from this one shortcoming, the model has proved general and flexible enough to express many common kinds of performance properties of interest to system implementors and users. We have seen examples of how to express assertions about response time, throughput, resource utilization, and workload characteristics. In each of these cases, we can express assertions about bounds and properties of distributions (such as means, variances, and percentages). In addition to being expressive, the model is simple and

applicable to many different kinds of systems, at many different levels of abstraction.

Chapter 3

The PSpec Language

The performance assertions we saw in the previous chapter were written using informal mathematical expressions and the notion of logs with events and intervals defined in the PSpec model. The PSpec language, which is based on the model, is a notation for writing performance assertions precisely so that they can be checked automatically and efficiently by machine. These assertions, contained in *performance specifications*, are input along with a monitoring log to the checker and solver tools, as described in Chapter 1.

Several questions had to be answered in designing the language:

- How are event types declared?
- How are interval types declared? In particular, how are the start and end events for an interval type identified and how are metrics defined?
- How are assertions written?

This chapter answers these questions, presenting the features of the language through a series of examples. A complete reference manual for the language appears in Appendix A.

As mentioned in the previous chapter, the PSpec language imposes restrictions on the model and on the allowable assertions to keep the language efficiently checkable and relatively simple. This chapter concludes with a discussion of the effects of the restrictions on the kinds of performance properties that can be expressed in the language.

3.1 Why a Special-Purpose Language?

Before considering the particulars of the PSpec language definition, it is reasonable to ask why a special-purpose specification language based on a

particular model of logs need be designed at all. After all, one can write programs to process logs in any general purpose programming language. There are at least three reasons why having a special-purpose language is a good idea. First, it is easier to write specifications. Programmers have had the ability to write log processing programs all along, and yet they do not do it very often. This is because the programs are tedious to write and without an established framework for log formats and log processing tools, the *ad hoc* log processing programs tend to get out of date with respect to the system. This means that they will not be used on a regular basis throughout the lifetime of the system, and they will not necessarily be understood by anyone but their implementors.

Second, specifications written in a special-purpose language will be easier to read. With severe discipline, one could perhaps write a C program in which it is clear what assertions are being made about the contents of a log. More likely, the C program would be much more difficult to read than a PSpec specification. This is because C, or any other general purpose programming language, has many concepts that are unnecessary for performance specification or are not really geared to the purpose. The reader gets bogged down in unnecessary details.

Third, for research purposes, it is easier to discover what concepts are useful for performance assertion checking if we can limit the concepts available to a specification writer. With a special-purpose specification language we are much more likely to learn what concepts work and what is really missing or incorrect than we would if the specification writer could just code up new abstractions or modify old ones using a general purpose language.

3.2 Overview

Here is a quick preview of how the questions posed in the introduction to this chapter have been answered.

An *event type declaration* gives the name of the event type, the names of its attributes, and indicates whether or not events of the type are *timed* (have implicit timestamp attributes). Timestamps are implicit so that the language can provide special functions that track measurement error. Events are restricted to have numeric-valued attributes.[1]

An *interval type declaration* gives the name of the interval type, indicates

[1] This is not a fundamental restriction, but rather was made for convenience in the current language design.

how to recognize instances of the type in a log, gives the names of its metrics, and says how to compute their values. Interval start and end events are identified primarily by their event type, and secondarily by a predicate on the values of the attributes of the event. There are two different kinds of interval declarations, to accommodate intervals generated recursively or non-recursively. Metric values are defined by expressions in the language that can use the start and end events for an interval, and that can iterate in restricted ways over the events and subintervals within the interval.

Assertions are predicates (boolean-valued expressions) in the language; they can include expressions that iterate in a restricted way over the events and intervals in a log, as well as expressions written using a variety of built-in operators.

A *performance specification* collects together a set of event and interval declarations and assertions, to be evaluated and checked against a monitoring log.

3.3 Language Description

We are now ready to see the features of the PSpec language in more detail. The examples used for illustration are based on the simplified file system described in the previous chapter. Along the way we will see how several of the assertions that were described informally in that chapter can be expressed precisely in the PSpec language.

3.3.1 Simple Declarations and Assertions

The simplest way to identify an interval type is to give the types of its start and end events, and the simplest kind of metric definition is one that uses only the values of the start and end event attributes. Suppose we want to define a simple *Read* interval type, corresponding to a file system Read operation, with a *time* metric whose value is the elapsed time for an interval of the type. We will ignore for the moment the need to match up *StartRead* and *EndRead* events for the same thread.

We start by declaring the event types of interest:

```
timed event  StartRead ();
             EndRead ().
```

These declarations define *StartRead* and *EndRead* to be timed event types with no attributes.

Now, we can define the interval type:

interval *Read* =
  s: StartRead,
  e: EndRead
metrics
  time = timestamp(e) − timestamp(s)
end *Read.*

This declares an interval of type *Read* to start with an event of type *StartRead* and to end with the next event of type *EndRead* after the start event. *s* and *e* are names for the start and end events for an interval of the type. Each interval of the type is declared to have a metric, *time*, whose value is obtained by subtracting the timestamp of its start event from the timestamp of its end event. (*timestamp* is a built-in function on timed events).

One simple kind of assertion is a predicate that applies to all events or intervals in a log. For example, given the *Read* interval type declaration, we can write an assertion about all intervals in the log of this type. Suppose we would like to say that "the elapsed time for any Read operation is at most ten milliseconds." We can do this with the following assertion:

assert {& *r : Read : r.time* ≤ *10 ms*}.

This assertion can be read as: "for all intervals *r* of type *Read*, the value of *r*'s *time* metric is at most ten milliseconds."

An assertion is simply a predicate in the language. The assertion above consists of a type of expression called an *aggregate expression* (it includes the braces and everything between them). An aggregate expression provides a way to generate a sequence of values and then combine them with an operator. In the aggregate expression above we generate a sequence of intervals by iterating over all intervals in the log of a specified type, *Read*, binding each in turn to the *dummy variable r*. The sequence of values to be combined with the operator & is the sequence of booleans resulting from evaluating the expression *r.time* ≤ *10 ms* for each value bound to *r*.

The *value expression* in an aggregate expression (the expression that appears after the second colon) may itself contain aggregate expressions, within limits. The rule is that an aggregate expression that iterates over events or intervals in a log, called a *log-aggregate* expression, may not refer to the dummy variables of outer aggregate expressions.[2]

[2]The qualification of "iterating over events or intervals in a log" is added because later we will see another kind of aggregate expression for which this rule does not apply.

Another simple kind of assertion computes a value using all events or intervals in a log and then asserts something about the value. For example, suppose that we would like to assert that "on average Reads take at most five milliseconds." The following assertion expresses this:

assert $\{mean\ r : Read : r.time\} \leq 5\ ms.$

This assertion contains an aggregate expression that uses the *mean* aggregate operator to compute the average value of the *time* metric for all intervals of type *Read*. It then checks that this mean is at most five milliseconds.

It is also worth mentioning here what the expressions *5 ms* means. The timestamps for events are values in some internal time unit. The expression *5 ms* means the number of internal time units equivalent to 5 milliseconds. Thus, *ms* is an operator that scales a number so that it is comparable to another number of internal time units. Other time unit operators are *cyc*, *us*, *sec*, and *min*, for cycles, microseconds, seconds, and minutes, respectively.

3.3.2 More on Identifying Intervals

So far, we can identify intervals that start with a particular type of event and end with the next event in the log of a particular type. Because this is not always expressive enough, the PSpec language gives a specification writer two additional ways of controlling interval identification. One is to allow start and end events to be constrained with a predicate in addition to a type. The other is to provide a variation on interval declarations that works for intervals corresponding to recursive computations.

Predicates on events. Predicates on start and end events give more control over how events are matched up to form intervals. For example, if multiple threads are doing Read operations, we need to match up *StartRead* and *EndRead* events for the same thread to form *Read* intervals. We can modify the interval type declaration to express this.

First, we add a *tid* (thread identifier) attribute to our events (also taking this opportunity to add a *size* attribute to *StartRead* events for later use):

timed event *StartRead (tid, size);*
EndRead (tid).

Given these definitions, we can alter the interval type declaration to specify that the thread identifiers of the start and end events of an interval must match:

```
interval Read =
    s: StartRead,
    e: EndRead where e.tid = s.tid
  metrics
    time = timestamp(e) − timestamp(s),
    size = s.size
  end Read.
```

Here we have added a *where clause* following the declaration of the end event type. The "where" clause contains a predicate that restricts the end event for a *Read* interval to have the same thread identifier as the start event for the interval. Using this declaration, an interval of type *Read* starts with an event of type *StartRead* and extends through the next event of type *EndRead* with the same *tid* as the start event.

In general, the predicate for a "where" clause following an end event may be any boolean-valued expression and may refer to the start and end events for the interval. The only restriction is that the predicate may not contain an aggregate expression.[3] A "where" clause may also be attached to a start event, in which case it can refer to the start event but not the end event. "Where" clauses on start events do not appear to be as useful as "where" clauses on end events, though one could imagine some uses. For example, if we wanted to define an interval type corresponding to Reads of a fixed size (say 8192 bytes), we could add a "where" clause to the start event:

```
s: StartRead where s.size = 8192.
```

Note that the start and end event types in an interval declaration may be the same. For example, if we wanted to write a workload assertion about the rate of Read requests from any thread (as in the previous chapter), we could define an interval:

```
interval ReadRequestInt =
    s: StartRead,
    e: StartRead where e.tid = s.tid
  metrics
    time = timestamp(e) − timestamp(s)
  end ReadRequestInt.
```

[3] It is not clear what an aggregate expression in a "where" clause for a start or end event should mean if it were permitted.

An interval of type *ReadRequestInt* starts at an event of type *StartRead* and extends to the next event of type *StartRead* with the same thread identifier. Thus a *StartRead* event can both start one interval and end another interval. However, there is no possibility that an event could start and end the same interval, since by definition the end event of an interval always follows the start event in the log. The last *StartRead* event in the log does not start a *ReadRequestInt* interval because there is no matching end event.

Recursively generated intervals. An interval has been defined thus far to have its end event be the next event in the log after its start event that has the specified type and that satisfies the "where" clause (if one is present). Consider though what would happen if we were generating start and end events for intervals corresponding to recursive procedures. The first end event generated for the innermost recursive call would be taken as the end event to match all previous unmatched start events for intervals of the type. This is clearly not what is intended. To handle this case, there is a variant of an interval type declaration called a *nested interval type*. A nested interval type is defined by including the keyword ***nested*** before the declaration. The end event for a nested interval i is defined be the next event in the log after i's start event that is of the specified type and that satisfies the "where" clause (if one is present), and in addition, that is not also the end event for another interval of the same type that started after i.

For example, suppose we wanted to define intervals corresponding to calls of a recursive depth-first search procedure. The declaration might be:

```
timed event  StartDFS (); EndDFS ();
nested interval  DFS =
     s: StartDFS, e: EndDFS;
  end DFS.
```

If we invoked the depth-first search procedure on a tree with a root and two leaf nodes, we would end up with three *DFS* intervals, one corresponding to the call on the root and one for each of the two leaves. The only difference between nested and non-nested interval declarations is how the end event is identified.

3.3.3 More on Expressing Assertions

We have seen how aggregate expressions allow us to compute functions over all events or intervals of a specified type in a log. Sometimes, however,

we would like to compute over a subset of events or intervals of a type. For example, suppose that instead of writing an assertion about all *Read* intervals, we would like to assert that "Reads of no more than 4096 bytes take at most ten milliseconds." We can express this assertion as follows:

assert $\{\& \; r$ *: Read* ***where*** $r.size \leq 4096 : r.time \leq 10\ ms\}$.

We are again using an aggregate expression, but this time we have added a "where" clause to restrict the range of the aggregation. Now, rather than checking the time bound for every interval of type *Read*, we only check those intervals where the size of the Read is at most 4096 bytes.

The "where" clause here is similar to the "where" clause that can follow interval start and end events. It contains a predicate that may refer to the dummy variable for the aggregate expression (r in this example). The "where" clause in an aggregate expression may itself contain aggregate expressions, though any contained log-aggregate expressions may not refer to the dummy variable.

3.3.4 More on Defining Metrics

Thus far, we have seen how to define a fixed number of metrics for an interval and to compute their values using the start and end events. Additional features in the language allow us to compute metric values using all of the events or subintervals within an interval, and to declare a collection of metrics whose number depends on the contents of the log.

Extending metric definitions. To allow metrics to be computed using all of the events or subintervals within an interval, the notion of aggregate expressions is extended and aggregates are allowed to appear in metric definitions. To illustrate, suppose that we would like to add a *hit* metric for *Read* intervals that is true whenever a Read hits entirely in the cache. As described in the previous chapter, we detect this case by the presence of a *CacheHit* event between the start and end events for a *Read* interval. Then we could, for example, use the metric to assert that the cache hit rate is at least seventy-five percent.

First, we declare the *CacheHit* event type, which can be untimed:

event *CacheHit (tid).*

Then we define the *hit* metric for *Read* intervals (which would be added to the earlier *Read* interval declaration):

$$hit = \{count\ c : CacheHit\ \textbf{where}\ c.tid = s.tid\} \neq 0.$$

The definition of *hit* uses aggregate expressions in a new way. Implicitly, when an aggregate expression appears in an assertion or other "top-level" expression in a specification, it ranges over all events or intervals in the log. We saw this type of aggregate expression in the previous examples. When an aggregate expression appears in a metric definition, however, it is defined to range over just the events or intervals that occur between the start and end events of the interval for which the metric is defined. Thus for each *Read* interval, the value of its *hit* metric will be based just on the *CacheHit* events between the *StartRead* and *EndRead* events for the interval. Also, this definition of *hit* uses the special aggregate operator *count* to count the number of *CacheHit* events in the interval with the same thread identifier as the start event. *Count* is special because it does not have a value expression after the range specification. It is really just a convenient shorthand for the expression $\{+\ c : CacheHit \ldots : 1\}$

We can now express the assertion about the cache hit rate for Reads:

$$\textbf{assert}\ \{count\ r : Read\ \textbf{where}\ r.hit\} / \{count\ r : Read\} \geq 0.75.$$

The expression in this assertion counts the number of *Read* intervals in the log for which *hit* is true, divides by the total number of *Read* intervals, and then checks whether the result is at least 0.75.

As mentioned above, aggregate expressions in metric definitions, like aggregate expressions in assertions, can range over intervals as well as events. What does this mean? An interval i is contained within another interval j if i's start and end events occur between j's start and end events. In this case we call i a *subinterval* of j. As an example of how subintervals can be useful, consider the example from the previous chapter of an assertion about the lock waiting time during a Read operation. Let us see how we can express the assertion that "the lock wait time during Read operations is less than one millisecond in at least ninety percent of the cases."

We define a *LockWait* interval that starts with a lock request event, ends with a lock granted event, and has a *time* metric giving the elapsed time for the interval:

```
timed event  LockRequest (tid);
             LockGrant (tid);
interval  LockWait =
    s: LockRequest,
```

```
    e: LockGrant where e.tid = s.tid
  metrics
    time = timestamp(e) − timestamp(s)
  end LockWait.
```

Then we can add a metric to the *Read* interval type that adds up the total time spent waiting for the lock during a Read:

$$lockwait = \{+\ l : LockWait\ \textbf{where}\ l.tid = s.tid : l.time\}.$$

Finally, we can write the assertion:

$$\textbf{assert}\ \ \{count\ r : Read\ \textbf{where}\ r.lockwait < 1\ ms\}\ /\ \{count\ r : Read\} \geq 0.9.$$

Aggregate expressions appearing inside interval type declarations (in metric definitions) may not contain nested aggregate expressions, either in the "where" clause or in the value expression—an important difference from aggregate expressions in assertions.

Dynamic numbers of metrics. The number of different named metrics for an interval is statically determined when a specification is written. Sometimes, though, we need a dynamic number of metrics. For example, we might want a metric per processor, or per thread, where the number of processors or threads is determined when a program runs rather than when the specification is written. A data type called a *mapping* gives us the ability to define a dynamic number of metrics. A mapping is a partial function from integers to values. The *domain* of a mapping is a set of integer values. A mapping associates each of its domain values with a *range* value of some type, not necessarily integer.

As an example of how mappings can be useful, let us consider a workload property discussed in the previous chapter—the average rate of Read requests to the file system on a per-thread basis. Suppose we would like to say that "the average Read request rate per thread is at most five requests per second." This translates to an average time between requests from a given thread of at least 0.2 seconds. Our plan is to use the *ReadRequestInt* interval type defined earlier to compute a metric whose value is a mapping from integer thread identifiers for threads that make Read requests to the average time between requests for the threads. Then we can write the desired assertion that applies over all elements of the mapping, using a new form of aggregate expression. The interval for which the metric is being

defined is, in fact, the whole log. The PSpec language has a special way to define metrics for the whole log without introducing an explicit interval type; a *def statement* introduces a name and binds it to the value of an expression.

Figure 3.1 summarizes the operations for manipulating mappings. The constructor, $\rightarrow$, forms a single element mapping whose single domain value is the value of its left argument (which must be an integer). This domain value is mapped to the value of the right argument. The indexing operation, *m(i)*, for mapping m and domain value i, returns the value to which i is mapped by m. It is an error if i is not in m's domain. The operation *mapped(m,i)* returns true if integer i is in m's domain, and false otherwise.

Operations are also provided for combining mappings. These are useful for building multi-element mappings from single-element mappings and for combining multi-element mappings. The result of applying a built-in operation *op* to a sequence of mappings is a new mapping whose domain is the union of the domains of the component mappings and whose value at a domain element i is the result of applying *op* to the values to which i is mapped in those component mappings that have i in their domains. For example, Figure 3.2 shows the result of combining two mappings, *m1* and *m2*, with the *mean* operator to produce a new mapping.

The definition of mapping combination is motivated by the intended use of mappings to represent a dynamic number of metrics. A domain value can be viewed as an identifier for a metric and the value to which it is mapped is the metric's value. Thus, combining mappings with different metric identifiers (disjoint domains) results in a new mapping containing all of the metrics of the component mappings. Combining mappings that share a metric identifier (domain value) results in a new mapping where that metric identifier is mapped to the combined values of the component metrics.

Getting back to our assertion, we can define the desired mapping from thread identifiers to the average time between Read requests for the threads as follows:

$$\textbf{\textit{def}}\ \ RMap = \{mean\ r : ReadRequestInt : r.tid \rightarrow r.time\}.$$

This statement binds the identifier *RMap* to the mapping constructed by the given aggregate expression. The aggregate expression constructs a sequence of single-element mappings, one per *ReadRequestInt* interval, each mapping the thread identifier for the interval to the elapsed time for the interval. These single element mappings are then combined with the *mean* operator,

e1 → e2:	construct a mapping
m(e1):	return *e1*'s mapped value
mapped(m, e1):	test whether *m* maps *e1*
op(m1, m2):	combine mappings *m1* and *m2*

Figure 3.1: Mapping operations. *e1* and e2 are expressions, and *m* is a mapping.

Let:
m1 = (1 → 100, 2 → 200)
m2 = (1 → 200, 3 → 100)
Then:
mean(m1, m2) = (1 → 150, 2 → 200, 3 → 100)

Figure 3.2: Example of combining mappings. *m1* and *m2* are mappings. *mean* is a built-in operator defined on mappings.

to produce a mapping whose domain is the set of thread identifiers appearing in *ReadRequestInt* intervals and whose value for a given thread identifier is the mean of the times of *ReadRequestInt* intervals for that thread.

Given this mapping, we can express the desired assertion using a special form of aggregate expression that iterates over the domain of a mapping rather than over a sequence of intervals or events.

assert {& *t* ***in*** *domain(RMap) : RMap(t)* ≥ *0.2 sec*}.

The range of iteration in this aggregate expression is the domain of the mapping bound to *RMap*. For each domain value *t*, the assertion checks that *RMap(t)* is at least 0.2 seconds.

3.3.5 Specifications

A *performance specification* is a document that pulls together a set of performance assertions with their accompanying declarations. A specification has the form:

perfspec *specid*
statements
end *specid*

where *specid* is an identifier that names the specification and *statements* is a list of event type declarations, interval type declarations, definitions, and assertions of the kind we have seen throughout this chapter. The specification name can be used to distinguish the event type names declared in one specification from those declared in another, if desired. (The connection between event type names in specifications and event types in logs is not prescribed by the PSpec language definition.)

3.3.6 Measurement Error

We have now discussed all of the important PSpec language features except one that does not directly have to do with the model. The one additional feature is a data type, called a *triple*, that is provided to help track measurement error resulting from quantization effects in the clocks used to generate timestamps.

Measurement error can occur when timestamps are recorded as a number of clock ticks, where the clock tick time is larger than the cycle time of the machine.[4] Typically, we measure the elapsed time to execute a piece of code by reading the number of clock ticks at the beginning and end, and taking the difference. This gives us only an approximation to the elapsed time: it is somewhere between one clock tick more than the difference of the readings and one less than the difference (assuming the measured time is non-zero), depending upon where in the tick interval we happened to read the clock.

If we compute an elapsed time for some computation of interest by adding up a collection of measurements, the potential error in the result must be taken into account if the data is to be interpreted properly. This is particularly important when the elapsed times being measured are only a small number of clock ticks in duration, because then the error is quite large relative to the measured time. An example of a consequence of ignoring measurement error when writing performance specifications is that a response time bounds assertion might fail because the measured time appears to be longer or shorter than expected, when in fact it is within the expected range to within measurement certainty.

[4] It is not unusual for currently available workstation systems to have clock tick intervals lasting as long as 10 milliseconds, whereas their cycle times are usually measured in nanoseconds.

The PSpec language provides triples as a way of keeping track of bounds on measurement error due to clock quantization effects. A triple contains a measured time interval and its associated error bounds, plus and minus. For example, the triple *[5,1,1]* represents a measurement of five ticks with errors of at most one tick in either direction. The triple *[0,1,0]* represents a measurement of zero ticks with a possible error of plus one tick; the error in the minus direction is zero, since we assume that the measured interval cannot have a negative duration. The language provides operators that construct triples and that compute with triples, described in detail in Appendix A. One very useful operator is the *elapsed* function on intervals, which returns a triple representing the elapsed time for an interval, computed from its start and end event timestamps.

One might argue, justifiably, that bounds on measurement error are not good enough. Intuitively, it seems that when we measure an elapsed time of t by differencing timestamps, not all values between $t-1$ and $t+1$ are equally likely values for the elapsed time; rather there is some non-uniform distribution of probability that the duration was between $t-1$ and $t+1$, with the most likely value being t. Under suitable assumptions about the independence of the clock phase and measurement points, we can carry out an analysis that shows this intuition is correct and determines the distribution. However, the assumptions are not clearly justified in practice and, even if they were, computing with distributions is not easy, while computing with bounds is. Lacking a better solution I chose to include triples in PSpec in preference to ignoring measurement error altogether. It is not yet clear whether the potential looseness of the bounds will be a problem in practice.

3.3.7 Summary of Features

To summarize, the features of the PSpec language are:

- Event type declarations: Events are the primitive components of logs. Event type declarations set up the connection between a log and a performance specification, giving the names of the event types in a log and the names of their associated attributes.

- Interval type declarations: Intervals are identified by giving the types of their start and end events and, optionally, predicates that must hold for the events. Metrics can be computed using all of the events and subintervals contained within an interval.

- Data types: The data types in the language are numbers, booleans, events, intervals, mappings and triples. Mappings provide the means for defining a dynamic number of metrics. Triples are three-tuples that are used to track bounds on measurement error resulting from clock quantization effects.

- Expressions: the language provides a set of built-in arithmetic, logical, and relational operators ($+$, $-$, $*$, $/$, *div*, *mod*, *abs*, *min*, *max*, *log*, *power*, &, |, !, $\Rightarrow$, $>$, $\geq$, $<$, $\leq$, $=$, $\neq$). Aggregate expressions are used for writing assertions and defining metrics. They provide a way of generating and combining sequences of values by iterating over events or intervals in a log, or over domains of mappings. A set of built-in aggregate operators is provided ($+$, $*$, &, |, *count*, *mean*, *var*, *stdev*, *min*, *max*, *the*, *last*, *first*). Other types of expressions include constructors and operators for mappings and triples.

3.4 Language Design Choices and Tradeoffs

The guiding philosophy of the PSpec design has been to keep the language small, including only features that are reasonably general and for which I saw a clear need in the examples I studied. The language and model evolved together over a period of about two years, going through two major versions. Some, but not all, of the examples discussed in this and the previous chapter had been examined at the time that the second versions were developed. New examples have motivated further ideas for language changes and model changes that have not yet been ironed out and incorporated into new versions. These ideas are discussed in detail in Chapter 6. The discussion for now will focus on the current versions of the language and model.

In addition to the high-level goal of producing a language for writing performance specifications based on the PSpec model, I had several subgoals in mind while designing the PSpec language:

- Most specifications should be efficiently check-able—check-ing them should take time linear in the length of the log, and the space required should be independent of the length of the log. For the cases where this is not true, the cost of language features should be transparent to a specification writer, who can then decide whether to pay for the added expressiveness.

- The language should be capable of expressing a wide variety of performance properties for many different kinds of systems.
- The specifications should be readable, so that they can serve as documentation, in addition to being checked.
- Efficient monitoring should be possible—the language should not necessitate putting redundant information in the monitoring log.

As usual, there are tensions among these goals, and tradeoffs must be made to arrive at a reasonable design. For example, efficiency of checking specifications must be traded off against expressive power. There are many predicates one could imagine that are not checkable in time linear in the number of events in a log—for example, some predicates of the form $\forall i : \forall j : P(i, j)$, where i and j are events and P is a predicate. As another example, expressive power can also conflict with readability. A general purpose programming language provides the power to process a log any way that we like, but it may not be readily apparent from reading the resulting program what performance assertions are being checked.

Given the general design goals, some of the important, specific language design issues were:

- The form of interval type declarations, in particular restricting interval starts and ends each to be identified by a single event type with subsidiary predicates;
- The design of aggregate expressions and their use for describing iterative computations;
- The facilities for describing how metric values are computed;
- The decision to make the language non-extensible.

I will discuss each of these issues in turn, explaining the impact of the choices on the success in meeting the design goals.

3.4.1 Intervals

The PSpec model defines an interval as a subsequence of a log—it starts at some point in the log, ends at a later point, and includes all events in between. The model does not say how these subsequences are identified—one could imagine many possibilities. A significant choice made in the PSpec

design was to identify intervals primarily by naming a single event type each for the start and end events. "Where" clauses then give some additional control over whether a candidate event (one of the correct type) is actually the start or end event for an interval.

There are at least two kinds of examples I have come across where the language restrictions on the form of interval type definitions make it difficult to write the assertions that we might want to write. The two cases are: identifying the start or end events for an interval type using a disjunction of event types, and identifying fixed-time intervals.

An example of where event type disjunction might be useful is an assertion about the lifetimes of files in the file system. Suppose we would like to assert that many files have short lifetimes, that is, they are either deleted or overwritten shortly after being created. We could imagine defining an interval type that corresponds to the time between when a file is created and when it is deleted or overwritten. Then we could write the assertion in terms of the elapsed times for all intervals of the type. But can we define such an interval type? The problem is that we must specify a single end event type when defining an interval type. Thus, we need an event type corresponding to a file being either overwritten or deleted. If we have events for each of these cases separately, there is no way to specify the end event type as the disjunction of these two types. It is not difficult to generate events for the "overwritten or deleted" case, but it means logging additional events that would not be needed if the language were more powerful.

An example of identifying fixed-time intervals was discussed in Chapter 2. We wanted to write assertions about average Read throughput over ten-minute intervals and the suggestion was to identify a *TenMinute* interval type either by using the existing timestamps on events or by generating explicit events in the log that mark off ten-minute intervals. Given the facilities provided by the PSpec language, only the latter option is available because there is no way to specify the interval type using just the event timestamps. Generating the extra events when timestamps already contain enough information means that the monitoring is less efficient than it could be. Also generating the events is inconvenient since we might want to change the duration of the interval in the specification without having to regenerate a log. Some means of identifying fixed-time intervals in the language would probably be a worthwhile addition.

There are other possible variations on interval identification that we can imagine but that have not yet seemed necessary based on experience. For example, we could imagine identifying intervals by predicates on the attributes

of the start and end events without having to specify a particular event type. Another possibility is to identify intervals by patterns of events rather than just by their endpoints. It is not clear whether either of these ideas could be implemented efficiently enough, and both would add complexity to the language. Following the design philosophy, I would not advocate adding them unless there was a clear need.

3.4.2 Aggregate Expressions

The PSpec language is assignment-free, a choice motivated by readability considerations and manifested primarily in the use of aggregate expressions to perform computations over sequences of events and intervals and over domain values of mappings. One alternative would have been to allow a specification writer to introduce state variables and to say how they are updated as each event is read from a log. Looking at a specification written in that style, it is often much less obvious what property is being computed for a log. Aggregate expressions are concise, readable, and reasonably powerful.

Aggregate expressions are restricted in two ways for efficiency reasons. All scalar aggregate operators use only an amount of space that is independent of the length of the log.[5] This space restriction rules out exact percentile operators, for example. So, we cannot express the assertion that the median time for Read operations is at most ten milliseconds. However, we can say that at least fifty percent of Read operations take less than ten milliseconds.[6] This restriction is not fundamental—aggregate operators that use more space could easily be added to the language. The user would be warned of the potential cost, which would be incurred only when an expensive operator is used.

The other restriction is that nested log-aggregate expressions cannot refer to dummy variables of outer aggregate expressions, because otherwise the checker might require time quadratic or greater in the length of the log. So we cannot write an assertion that doubly iterates over all events or intervals in the log directly (although we could produce this computation indirectly, using mappings). We saw an example where such double iteration might have been useful (on page 31), but other solutions were possible and even preferable in that situation.

[5] A specification writer can, however, write an aggregate expression that produces a mapping with size proportional to the length of the log.

[6] It would also be possible to provide an approximate percentile operator that divides the data into a fixed number of bins and thus uses constant space.

3.4.3 Metric Definitions

Metric definitions are restricted in two ways so that the checker can recognize intervals in a log using a single pass over the log file and a reasonable amount of space. One of the restrictions is that expressions in metric definitions may not contain nested aggregates, even if the aggregates obey the rules about referencing outside dummy variables. The other restriction is that aggregate expressions in metric definitions may not refer to the end event for an interval. The reasons for these restrictions should be more intelligible after reading the description of the implementation strategy in Chapter 4. In terms of expressiveness, these restrictions on metric definitions have not posed any difficulties in the examples that I have studied to date.

3.4.4 Non-extensibility

The set of operators in the language is limited to the built-in operators. This choice was made for two reasons. One is that the focus of the research was on performance assertion checking, not programming language design. The design of a mechanism for user-defined operators would not have shed any light on how to write or use performance specifications. The second reason relates to the reason for designing a special-purpose language in the first place—that it is easier to discover whether a feature is necessary by leaving it out to begin with. It is not clear how important extensibility is in this context. If it proves to be important it could be added without serious impact on the rest of the language.

One example of where lambda abstraction would probably be useful is for complicated expressions involving events or intervals. For example, when working with elapsed time models for Read operations in Chapter 2 we had the expression:

$$PerByteTime \times r.size + Overhead + r.wait.$$

Our assertions might be more readable if we could write them having defined a function *expected(r)* that is parameterized on an interval of type *Read* and returns the value of the above expression when applied to an interval. Perhaps we could write something like:

def *expected(r: Read)* $= PerByteTime \times r.size + Overhead + r.wait;$
assert $\{\& \; r : Read : r.time \leq expected(r)\}.$

Such an extension to the language would probably be worthwhile.

Chapter 4

Tools

This chapter describes the functionality and implementation of the tools provided for performance assertion checking. The *checker* takes a performance specification and a monitoring log and reports which assertions in the specification fail to hold for the log. The *solver* takes a performance specification with *unknown* constants and a monitoring log and estimates values for the constants. Both of these tools must read a performance specification, recognize the intervals defined for a log, and evaluate expressions relative to a log. After describing the functionality of the checker and solver I describe the implementation of the specification parsing and evaluation library common to both tools, and analyze the time and space required for evaluation. The analysis clarifies the conditions under which efficient specification evaluation is possible. The chapter concludes with a brief discussion of ideas for several other tools that could be useful for performance assertion checking; these other tools have not been designed.

4.1 Checker

It is the checker's job to report when the predicates occurring in assert statements in a performance specification, evaluated relative to a given monitoring log, are false. The current checker works off-line. The system being monitored is run for as long as necessary to gather a complete log against which a specification can be checked. Then the log can be checked at any time. This design works well for the setting in which the checker has been used thus far (described in Chapter 5). One could imagine checking specifications on-line, as data is generated from a running system. Such designs are possible, but have not yet been explored in detail. Chapter 6 discusses some of the issues.

The checker outputs two forms of information: messages reporting which assertions failed, and debugging information that can help as a first step in tracking down the causes of the failures.

There are two basic forms of assertion failure messages. One simply reports that an assertion failed and gives the line number in the specification identifying which assertion it was. The other form reports the line number and, in addition, the number and identity of the events or intervals in the log that cause the assertion to be false. This second form is used for assertions whose outermost expressions are log-aggregates using the conjunction operator. In such cases, the particular intervals or events for which the predicate inside the aggregate expression evaluates to false are identified as causing the assertion failure.

For example, suppose we are checking the assertion:

assert $\{\&\ r : Read : r.time \leq 1\ sec\}.$

If there were *Read* intervals that had elapsed times of greater than one second, a message would be produced indicating that the assertion failed and identifying the offending intervals. The intervals are identified with reference to the debugging information also produced by the checker. An example of an assertion for which it is not possible to identify particular intervals that cause a failure is:

assert $\{mean\ r : Read : r.time\} \leq 500\ ms.$

The debugging information produced by the checker consists of an entry for each interval in the log. An entry contains: a unique identifier for the interval that is used to refer to the interval in assertion failure messages, the interval type, the location of the interval's start and end events in the log and their timestamps (if any), and the values for the interval's metrics. This information can be examined with a text editor or other text processing program.

This form of output is somewhat crude, but provides a rudimentary capability to begin tracking down the causes of assertion failures. Examining the values of metrics may give clues as to what was going on in the system when the performance was not as expected, and which parts of the system may be responsible. Chapter 5 describes the experience I had using the checker to track down performance bugs. Better tools are needed for examining logs and relating them to program code. Some ideas for these are discussed in Section 4.4.

4.2 Solver

The solver helps to answer the question of how to fill in values of constants in performance specifications. For example, in Chapter 2, in discussing assertions about models for elapsed times of Read operations we had two constants, *PerByteTime* and *Overhead*, whose values would have to be supplied in order to check the specification. Sometimes a specification writer knows the values for constants *a priori*, perhaps because they reflect particular performance goals. Often though, the implementors of a system will tune the system's performance as well as possible and then conduct experiments to determine the values of performance constants. These experiments typically involve measuring the system, computing metrics, and performing other calculations such as curve-fitting. Measurement and metric computation are activities that are also associated with performance assertion checking. The solver provides equation-solving capabilities in the PSpec framework.

4.2.1 Overview

The input to the solver is a monitoring log and a performance specification containing *unknowns* and *solve declarations* (in addition to the usual event and interval type declarations, definitions, and assertions). An unknown is a symbolic constant (such as *PerByteTime*) whose value is left undetermined by the user. A solve declaration is a directive to the solver describing how to estimate values for unknowns.

The output of the solver is the revised specification, with unknowns instantiated with their estimates. This specification can then either be used as is, or modified by a specification writer, to check against other monitoring logs. Typically, the monitoring log given to the solver would be from a run specially designed to produce good estimates for the unknowns.

An unknown is identified using an alternative form of def statement in a specification. The statement:

def *id* = *?*

declares *id* to be an unknown.

Solve declarations come in two forms. The simple form contains an equation that is linear in one unknown. The equation can include expressions that must be evaluated relative to a monitoring log. The solver simply computes the value of the unknown using algebra. The second form of solve declaration describes how to compute a set of data points and gives an

equation for use in a linear regression analysis that produces estimates of values for the unknowns.

The two examples below should give a feel for how the solver is used in the two situations. These examples are based on the simplified file system from the preceding chapters. In particular, we will look at models for the elapsed time of Read operations. Earlier we hypothesized that the elapsed time for a Read operation r that hits in the cache is given by the formula:

$$r.time = PerByteTime \times r.size + Overhead + r.wait.$$

For simplicity, we will assume that the lock waiting time, *r.wait*, is zero here. (If it were non-zero, we could simply subtract it from *r.time* to solve the equation.)

4.2.2 Easy Example

We begin by considering how we can use the solver to find the value of an unknown appearing in an equation linear in that unknown. Suppose that we would like to estimate the mean elapsed time for a file system Read operation of a fixed size (say 8 kilobytes) that hits in the cache. Since our model hypothesizes that the elapsed time varies only with the size of the Read, which we have fixed for now, the model predicts that all 8-kilobyte Reads should have the same elapsed time. In fact, our model is probably not entirely accurate, and there will be some slight variation in the time for 8-kilobyte Reads, but we expect this variation to be small. It therefore makes sense to compute the mean time for 8-kilobyte Reads and write an assertion saying that the actual time for an 8-kilobyte Read is in some reasonable range of the computed mean.

We can instruct the solver to compute the mean elapsed time for 8-kilobyte Reads as follows. First, we introduce an unknown for this elapsed time:

def *Mean8kRead = ?.*

Then we write a solve declaration containing an equation that uses the unknown, indicating how to estimate its value.

solve *Mean8kRead =*
{mean r : Read ***where*** *r.size = 8192 & r.hit : r.time}.*

Recall that the aggregate expression in braces is read as: "the mean, over all *Read* intervals r for which $r.size$ is 8192 and $r.hit$ is true, of the values of $r.time$."

Given, in addition, a log containing many 8-kilobyte Read operations, the solver can compute the value of the right-hand side of the equation to obtain the estimate for *Mean8kRead*.

Once we have the estimated value for *Mean8kRead* we can, for example, write an assertion to check that the mean in any new log is within a reasonable range of *Mean8kRead*. To check that the computed mean is within five percent of *Mean8kRead* we could write:

$$\begin{aligned}
&\textbf{\textit{def}}\ \ Mean8kReadLow = Mean8kRead \times 0.95;\\
&\qquad Mean8kReadHigh = Mean8kRead \times 1.05;\\
&\textbf{\textit{assert}}\ \ Mean8kReadLow\\
&\qquad\qquad \leq \{mean\ r : Read\ \textbf{\textit{where}}\ r.size = 8192\ \&\ r.hit : r.time\}\\
&\qquad\qquad \leq Mean8kReadHigh.
\end{aligned}$$

4.2.3 Harder Example

Suppose now that we would like to estimate values for the constants *PerByteTime* and *Overhead* in the elapsed time model for Reads of any size that hit in the cache. One way to estimate these values is to conduct an experiment where we perform a set of Read operations of various different sizes and measure the elapsed time for each one. For each Read we get one data point consisting of the size and the measured elapsed time. Then, because our elapsed time model is an equation for a line, we can fit a line to these data points to get estimates for the unknowns; the slope of the line gives an estimate for the per-byte time and the y-intercept gives an estimate for the overhead (as illustrated in Figure 4.1).

To instruct the solver to do line-fitting, another form of solve declaration is used. First, we must declare the unknowns:

$$\textbf{\textit{def}}\ \ PerByteTime = ?;\ Overhead = ?.$$

Then the solve declaration tells the solver how to generate the data points and gives the equation to be solved. For this example, the solve declaration would be:

$$\begin{aligned}
&\textbf{\textit{solve}}\ \ \textbf{\textit{data}}\ r : Read\ \textbf{\textit{where}}\ r.hit :\\
&\qquad r.time = PerByteTime \times r.size + Overhead.
\end{aligned}$$

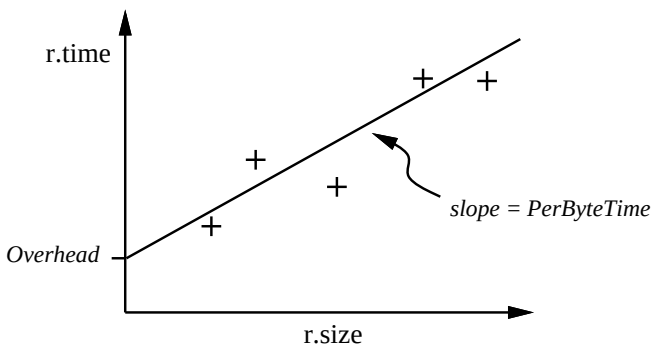

Figure 4.1: Estimating *PerByteTime* and *Overhead* by line-fitting

The first part, following the keyword *data* and ending with the second colon, tells the solver how to generate the data points. It is much like the range specification for an aggregate expression; it has a dummy variable, and a specification of a sequence of events or intervals in the log.[1] In this example, we are instructing the solver to iterate over all *Read* intervals whose *hit* metrics have the value *true*. Following this range expression is the equation to solve. The equation includes terms that refer to the dummy variable and to the two unknowns. For each event or interval in the range, the terms containing the dummy variable are evaluated to produce one data point.

The process used by the solver to fit a line to data is called *linear regression*. As noted earlier, the elapsed time model for Read operations is not exact. We expect some variation in actual measured elapsed times for Reads of any given size. A linear regression analysis makes certain assumptions about this variation. In particular, the assumption is that the model actually has the form:

$$r.time = PerByteTime \times r.size + Overhead + \epsilon$$

where ϵ is an error term that accounts for any unexplained variation in the elapsed time due to the inaccuracies of the model. ϵ is assumed to have a normal distribution with a mean of zero and some unknown variance V. Because for any Read operation r, $r.size$ is constant, and $PerByteTime$ and

[1] Range specifications that use the domains of mappings rather than events or intervals are also permitted, as for aggregate expressions.

Overhead are assumed to be constants, *r.time* is also normally distributed, with mean $PerByteTime \times r.size + Overhead$ and variance V.

In addition to producing estimates for the unknowns in an equation, linear regression analysis also produces an estimate for the unknown variance, V, and for a quantity called the *coefficient of correlation of regression*, which is a measure of how well the data fits a linear model. The user can obtain these quantities using additional clauses in the solve declaration.

Given these estimates, we can proceed to write assertions about the elapsed time for Reads. For example, we could assert that over all *Read* intervals in a log, the mean of the differences between the measured elapsed time for each operation and the time predicted by the model is close to zero (say, within one millisecond). We could also assert that the variance of that quantity is close to V.

$$
\begin{aligned}
&\textbf{\textit{assert}}\ \ -1\ ms \leq \\
&\qquad \{mean\ r : Read : r.time - PerByteTime \times r.size + Overhead\} \\
&\qquad \leq 1\ ms; \\
&\textbf{\textit{assert}}\ \ V - 1\ ms \leq \\
&\qquad \{var\ r : Read : r.time - PerByteTime \times r.size + Overhead\} \\
&\qquad \leq V + 1\ ms;
\end{aligned}
$$

4.2.4 Using the Solver Intelligently

As with many tools, the solver is only an aid if the user has a good understanding of its capabilities and limitations. The cliché "garbage in, garbage out" has particular relevance here. In order to get good estimates for unknowns out of the solver, the user must provide a log that contains data appropriate for producing those estimates and a specification that instructs the solver to compute the estimates in a reasonable way. This is not difficult if care is taken to understand how the solver uses specifications to compute estimates, and if the data and resulting estimates are examined for plausibility. In particular the user should:

- ensure that there are enough data points for averages to be meaningful;
- ensure that the range of supplied values for variables in the equations (such as the size of *Reads* in our example above) reflects the range expected in the logs against which the specification will be checked;
- check the coefficient of correlation to see whether a linear model is in fact appropriate for the data. Using additional techniques to check

the appropriateness of the model (such as plotting the data) is also a good idea.

The solver does not save the user the job of thinking—it simply mechanizes some tasks that might otherwise be tedious.

The linear regression capabilities of the solver will be most useful to someone who understands something about the uses and limitations of the technique. Most introductory statistics texts include a discussion of simple linear regression that should be helpful (see, e.g., [17]). Another commonly available reference is [39], which has some discussion of simple linear regression and multiple regression, including the formulas used to carry out a regression analysis and compute the coefficient of correlation (unfortunately it includes almost no motivation for the technique).

While this is not the place for a full discussion of linear regression, it is worth noting that the linear equation for the regression analysis need only be linear in the *unknowns*. For example, if we have an interval whose elapsed time is an unknown constant multiplied by the logarithm of one of the interval's metrics, this is still linear in the unknown and thus amenable to solution by the solver. Also, sometimes it is easy to transform an equation that is not linear in an unknown into one that is. For example, the equation "$y = x^a$", where a is an unknown, is not linear in a. However, "$\log y = a \log x$" is linear in a. Thus the solver can handle many more cases than might immediately be obvious.[2] But the solver does not perform these transformations automatically—the user must do it.

Even when the solver is being used for simple algebra the user should ensure that the data in the log is meaningful. For example, we wrote a solve declaration earlier to determine the mean elapsed time of fixed-size Reads, expecting that the elapsed times for individual Reads would not vary by much. It would also be worthwhile in that situation to compute the variance of the elapsed times and check that it is relatively small as expected. If it is not, that would indicate a flaw in our model.

4.3 Implementation

We now turn to the implementation of the tools, with a description of the implementation techniques employed for the specification parsing and eval-

[2] But the variance and coefficient of correlation computed by the solver would apply to the transformed equation, not to the original equation, so caution is required in interpreting the output.

uation library shared by the checker and solver.

4.3.1 Parsing and Type Checking

The first phase of interpreting a specification involves parsing, type inferencing, and type checking. Type inferencing involves choosing instances of overloaded operators based on the types of their arguments and inferring the types of expressions based on the types of the operators and constituent expressions. Type checking ensures that arguments to operators have the appropriate types and that expressions have boolean values where necessary (e.g., in "where" clauses and assert statements). All of this is accomplished in one pass through the specification. The result is a parse tree representing the specification that can be used in the evaluation phase.

4.3.2 Evaluation

The set of interval type definitions in a specification induces a partially-ordered set of intervals and events on a log of events.[3] Using this partial order, we can interpret a log as a sequence of events and intervals, making arbitrary ordering choices where the partial order does not determine the result. This abstraction of a log is called an *interval log*, whereas the primitive log is called an *event log*. The interval log abstraction is helpful for evaluating aggregate expressions in a specification (recall that aggregate expressions are the only kinds of expressions that depend upon the contents of a log). Once the aggregate expressions in a specification are evaluated, the other expressions (not dependent upon the log) can be evaluated.

Interval Log Stream. An event or interval in an interval log is a record containing a set of named fields with values (the attributes of an event, or the metrics of an interval). An event has, in addition, an index into the underlying event log and, possibly, a timestamp. An interval has indices for its start and end events in the underlying event log, and possibly, timestamps for those events. A sequence of events and intervals induced by a specification on an event log can be produced in a single sequential pass through the event log.

The data structures used to implement the interval log abstraction are:

[3] Recall that an interval is ordered relative to other intervals based on its end event. An interval is unordered with respect to its own end event and any other intervals with the same end event.

- *inttypes:* the set of interval type definitions in a specification; obtained from a specification and fixed for the algorithm;

- *openints:* the set of *open intervals* (potential intervals whose start events have been encountered but whose end events have not been—and may never be—found); initially empty;

- *logaggrs:* the set of log-aggregate expressions[4] in metric definitions of all open intervals in *openints*, along with storage for incrementally evaluating these expressions; initially empty;

- *newlyclosed:* the set of intervals closed by an event; initially empty.

Figure 4.2 sketches an algorithm that uses these data structures to produce a sequence of events and intervals in an interval log corresponding to a given specification and event log. The statement *yield* x in the description indicates that event or interval x in the interval log stream is produced at that point. The algorithm loops through the sequence of events in the event log, yielding each event as it occurs, and also checking whether the event opens any new intervals, closes any open intervals, or applies to any aggregate expressions in metric definitions of open intervals. Each time an open interval is closed by an event it is also yielded as an element of the interval log stream, and it is applied to aggregate expressions for open intervals.

An event e *closes* an open interval o if e's type is the end event type named in o's type definition, and if e satisfies the end "where" clause (if there is one). Furthermore, if o has a nested interval type then e only closes o if o has a greater start event index than any other open interval of its type for which e satisfies the end "where" clause. e *opens* an interval of type t if e's type is the start event type named in t and if e satisfies the start "where" clause (if any).

An event or interval ei can be *applied* to a log-aggregate expression if it has the type named in the range clause of the aggregate expression and if it satisfies the "where" clause in the expression (if any). In addition, if ei is an interval, it only applies to an aggregate expression in a metric definition if ei's start event index is greater than the start event index for the interval containing the metric.

[4]Recall that a *log-aggregate* expression is an aggregate expression whose range is events or intervals in a log. These are distinguished from *mapping aggregates*, which range over the domains of mappings.

Algorithm for generating events and intervals in an interval log stream:

```
1. While there is a next event e in the event log do
2.     Yield e
3.     newlyclosed := {}
4.     For each open interval o in openints do
5.         If e closes o then
6.             Finish evaluating expressions in o's metrics
7.             Remove o from openints
8.             Remove o's log-aggregate expressions from logaggrs
9.             Add interval for o to newlyclosed
10.            Yield interval for o
11.        End
12.    End
13.    For each log-aggregate expression a in logaggrs do
14.        Apply e to a
15.        For each interval i in newlyclosed do
16.            Apply i to a
17.        End
18.    End
19.    For each interval type t in inttypes do
20.        If e starts an interval of type t then
21.            Create an open interval o and add it to openints
22.            Add o's log-aggregate expressions to logaggrs
23.        End
24.    End
25. End
```

Figure 4.2: Algorithm for generating interval log stream from an event log stream and a specification with interval type definitions.

Because log-aggregate expressions in metric definitions cannot themselves contain log-aggregate expressions, they can be evaluated incrementally in a single pass through the log. When a new open interval is created, storage is allocated for the incremental evaluation of the log-aggregate expressions appearing in the interval's metric definitions. During the scan of the log, whenever an event or interval applies to one of the log-aggregate expressions of an open interval, the value-expression of the aggregate is evaluated and the result is incorporated into the accumulated value stored with the open interval.

Now let us consider the execution time and space for the algorithm in Figure 4.2. The analysis ignores mappings. Let:

$$\begin{array}{rcl} v & = & \max_{e \in \text{ event log}}\{\text{number of intervals that contain } e\} \\ p & = & \text{size of specification} \\ n & = & \text{number of events in the event log} \\ m & = & \text{number of intervals in the interval log.} \end{array}$$

v is a measure of the maximum overlap among intervals in the interval log (the maximum number of open intervals in *openints*). p is a loose upper bound for a number of measures, such as the maximum number of log-aggregate expressions in any interval definition, the number of interval type definitions in a specification, and the maximum number of operators in an expression.[5] It will serve our purposes for this analysis since it is independent of both the number of events in the event log and the maximum overlap.

The space taken by the algorithm is $O(vp)$, obtained as follows. The number of elements in each of the sets *openints* and *newlyclosed* is $O(v)$, and each element takes space $O(p)$, so these sets require space $O(vp)$. The set *logaggrs* also requires space $O(vp)$ because the log-aggregate expressions for each open interval take space $O(p)$ and there are $O(v)$ open intervals. The set *inttypes* requires space $O(p)$. Thus the total space required is $O(vp)$.

The time taken by the algorithm is $O(vp^2(m+n))$, obtained as follows. The time to execute lines 5–10 is $O(p)$, and they are executed $O(v)$ times for each of the n events. Thus lines 5 through 10 take time $O(vpn)$. Line 16 takes time $O(p)$, and is executed $O(vpm)$ times for the entire algorithm, for a total of $O(vp^2m)$. Line 14 takes time $O(p)$ and is executed $O(vp)$ times for each of the n events, giving a total time of $O(vp^2n)$. Lines 20–22 take $O(p)$ time and are executed $O(p)$ times for each of the n events, for a total of $O(p^2n)$. Adding this all up, we get we get $O(vp^2(m+n))$.

[5] Thus, the time to evaluate an expression or to apply an event or interval to an aggregate expression is $O(p)$.

Evaluating Top-Level Expressions. Given the interval log, the rest of the evaluation process involves evaluating expressions in definitions and either in assertions or in solve declarations (depending upon whether the evaluation is being done by the checker or the solver). Any of these statements can contain log-aggregate expressions that must be evaluated relative to a log, as well as other expressions that can be evaluated once all contained log-aggregate expressions have been evaluated. Log-aggregate expressions contained in any of the types of statements listed above are called *top-level*, meaning that they range over the entire log. A top-level log-aggregate expression may still be nested within other top-level log-aggregates.

The strategy for evaluating expressions in def, assert, and solve statements is to evaluate all log-aggregate expressions and def statements as soon as possible, and then to evaluate all other expressions. A log-aggregate expression can be evaluated as soon as all contained log-aggregate expressions have been evaluated. Because of language restrictions on log-aggregates (in particular, that a log-aggregate expression cannot reference dummy variables of outer aggregate expressions) it is always possible to evaluate an inner log-aggregate expression before evaluating any expressions that contain it.

We say that a log-aggregate expression *has depth 1* if it contains no log-aggregate expressions. A log-aggregate whose contained log-aggregates have at most depth 1 has depth 2, and so on. We also classify def statements into depths according to the maximum depth of their contained log-aggregate expressions. Defs with no contained log-aggregates have depth 0, defs with depth-1 log-aggregates have depth 1, *etc*. The algorithm for evaluating specifications, using the interval log abstraction described earlier, and assuming that the log-aggregate expressions in def, solve, and assert statements have depth at most r, is sketched in Figure 4.3.

A solve declaration that uses linear regression is treated as a special kind of log-aggregate expression (with depth one greater than the depth of its contained log-aggregates) during the iteration through the interval log shown in Figure 4.3.

The space taken by the specification evaluation algorithm in Figure 4.3 is $O(vp)$ as for the interval log generation algorithm since the additional space needed for evaluating top-level log-aggregates and other expressions is $O(p)$.

The time to execute the entire evaluation algorithm (including recognizing intervals) is $O((r+v)p^2(m+n))$, computed as follows. The time to execute the loop bodies at lines 11 and 14 is $O(p)$. These loops (at lines

Algorithm for evaluating specifications:

1. For each depth-0 def statement d do
2. Evaluate d
3. End
4. For $i := 1$ to r do
5. For each event or interval ei in the interval log do
6. For each depth-i log-aggregate a do
7. Apply ei to a
8. End
9. End
10. For each depth-i log-aggregate a do
11. Finish evaluating a
12. End
13. For each depth-i def statement d do
14. Evaluate d
15. End
16. End
17. For each assertion or solve statement s do
18. Evaluate s
19. End

Figure 4.3: Algorithm for evaluating specifications. r is the maximum depth of nesting for log-aggregates in def, assert, and solve statements.

10 and 13) each run for $O(p)$ iterations, for each of the r values of i. Thus the total time to execute these loops is $O(rp^2)$. The loops at lines 1 and 17 each execute for $O(p)$ iterations, with time $O(p)$ for each of the bodies, giving a total time of $O(p^2)$. The loop at line 6 again takes time $O(p^2)$, and is executed $O(r(m+n))$ times, for a total time of $O(rp^2(m+n))$. Also, as determined earlier, it takes $O(vp^2(m+n))$ time to generate all the values for ei on line 5 for each of the r values of i. The total is thus $O((r+v)p^2(m+n))$ for the entire evaluation.

Now let us consider the implications of a running time of $O((r+v)p^2(m+n))$. The total number of intervals m is $O(np)$ because the number of intervals started by each event is at most the number of interval type definitions. Also, r is $O(p)$. Thus the running time can be expressed as $O(vp^3n + p^4n)$. p is independent of, and likely very much smaller than, n. Also, recall that p is a very loose upper bound on the actual parameters of interest for the analysis. So the factors of p^3 and p^4 are not particularly worrisome. If v is independent of, and much smaller than, n, we have an algorithm whose running time is nearly linear in the length of the log (depending also on v and p, but dominated by n). In the worst case, however, v can be $O(n)$, making the algorithm quadratic in the length of the log. This could happen, for example, if every event starts an interval that ends with the last event in the log. It could also happen in a program that is deeply recursive (e.g., a recursive traversal of a list). For very long logs, a checker that takes time quadratic in the length of the log will have intolerable performance. The specification writer must take this into account when defining interval types for a program, and try to arrange that v is not too large.

As mentioned earlier, the analysis ignores mappings. This is reflected in the assumption that expressions can be evaluated in time $O(p)$, as well as in the space analysis. A mapping could have size proportional to the length of the log, causing the evaluation of an aggregate expression over the mapping to take time $O(np)$. In this case, the evaluation algorithm could take time quadratic or greater in the length of the log. The specification writer must also be aware of this possibility and use mappings judiciously.

4.4 Other Tools: Wish List

In addition to the checker and solver, there are a variety of other tools that one can imagine, both to provide better support for performance assertion checking and to provide additional functionality within the performance specification framework.

Evaluator. An evaluator tool would take as input a performance specification and a log. It would provide an interactive read-eval-print loop for evaluating expressions relative to the log, using the events and intervals declared in the specification. Such a tool would be useful for helping to understand assertion failures for a log, as well as for gaining understanding about the performance of a system before writing assertions in the first place. Perhaps the tool would also allow the user to declare new interval types interactively.

Visual display. Another kind of tool that would be useful both for understanding assertion failures and for understanding system performance in general would create graphs or other visual displays of data, taking a log and specification as input. Graphs could be defined in terms of intervals and metrics from the specification, interpreted relative to the log.

Log debugging. Generating correct monitoring logs is not always easy to do, as anyone who has tried probably knows. Even assuming that the code implementing the mechanics of event logging is not buggy, it is still very easy to end up with missing events because the person doing the instrumentation forgot to put logging calls into the source in some places. For this reason, some sorts of log debugging tools will be required. At a minimum, it is probably necessary to be able to get a human-readable dump of the log events, so that the specification writer can see the log contents. Tools that give additional assistance in examining a log—perhaps by filtering it in various ways so that subsequences can be examined—will be very valuable.

In addition to getting the monitoring right, the task of defining intervals on logs for use in specifications introduces additional opportunity for error. Intervals are a dynamic notion (they depend upon the execution path taken by the program) and it is easy for a specification writer to have a misunderstanding about what the sequence of events in the log will actually look like. For example, if a specification writer forgot to use thread identifiers to match interval start and end events in a case where there are in fact multiple threads generating the same event types to the log, then the intervals recognized by the checker based on the specification would not correspond to what the specification writer intended. Log debugging tools could also help to locate problems of this kind by helping the specification writer to understand what is in the log.

Chapter 5

Experience

From its inception this research has been motivated and guided by experience with the performance of actual systems. The ideas for the PSpec language came from studying the question of what kinds of performance properties were both useful for describing performance expectations and amenable to checking against monitoring data. This chapter describes an experiment using the initial version of the PSpec language and tools to write and check performance specifications for the runtime system of *Prelude*, a new parallel programming language [41, 42]. The experiment had two beneficial results. One was that some of the examples exposed shortcomings of the PSpec language and produced insights about how to generalize it. The other outcome was the discovery of performance bugs in the Prelude runtime as a result of using the PSpec tools.

After describing the Prelude experiments and evaluating the PSpec approach based on that experience, I will discuss the general methodology of the PSpec approach and how the intended use of the tools affects the kinds of performance specifications written.

5.1 Testbed

Prelude, a language for writing portable and modular parallel programs, is being developed by the Large-Scale Parallel Software Group at MIT. Prelude provides a programmer with a computational model based on objects and threads that abstracts away from the underlying architecture of the machine. High-level directives (pragmas) that specify the mapping of a program onto a particular architecture are added on top of the computational model to give a programmer control over the program's performance. Efficiently mapping a parallel program onto a machine involves choosing grain sizes of tasks,

determining where to place tasks and data, determining when and where to migrate tasks and data, scheduling tasks, managing communication among tasks, and determining how to cache, replicate, and partition data. Prelude takes the approach of allowing the programmer to provide directives to the compiler and runtime system, which then is responsible for the details of data structure layout and task decomposition.

At the time that I conducted the experiments, an initial version of Prelude had been designed and mostly implemented on top of *Proteus*, a high-performance simulator for MIMD architectures [6, 7]. Proteus executes programs written in a superset of C that has special facilities for handling shared-memory accesses, thread management, message passing, and synchronization. When the Prelude implementation is complete, Proteus will also be able to execute Prelude programs. Proteus provides a particularly nice environment for developing parallel software. Features include non-intrusive debugging (debugging or monitoring code usually can be added without affecting the timing of a simulation), repeatability, and integrated tools for data collection and display.

The Prelude/Proteus environment provided a good testbed for my performance specification experiments for several reasons. First, the people who designed and implemented the language were available and cooperative. The specifications were written based on the implementors' descriptions of their performance expectations. I served as the expert on the PSpec tools, explaining what could and could not be expressed in the language, translating their English descriptions of performance expectations into PSpec, and annotating the code to produce monitoring logs suitable for use by the checker.

Second, good performance in Prelude is necessary for its success. For the most part, the complexity of writing parallel programs is justified only by the improved performance they can deliver compared to sequential programs. The implementors tried to pay careful attention to performance (within the constraints of completing a prototype implementation in a timely fashion). They had definite performance expectations and were interested to see whether their expectations were met.

Third, it was easy to put the instrumentation into place to produce monitoring logs. Proteus already had a facility for generating event logs so it was merely a matter of producing some simple tools to connect names in PSpec specifications to names in logs and to provide some macros for use in annotating the runtime code to produce suitable events.

Finally, working on top of a simulator allowed me to avoid issues related

to efficient monitoring (since monitoring can be zero-cost in Proteus) and focus the research on the content of specifications and the form of the specification language. Monitoring issues are important and must be studied in order for the PSpec approach to be useful for non-simulated systems, but are separable from questions of what properties ought to be expressible in performance specifications.

5.2 Prelude Performance Specifications

With the help of the Prelude implementors I wrote performance specifications for several pieces of the Prelude runtime system. For a few of the specifications we also constructed test programs to exercise the corresponding pieces of the system, generated monitoring data, and checked the specifications. The others were not checked for two reasons. One is that not all pieces of the system for which we wrote performance specifications had been implemented at that time. The other reason is that I had begun to redesign the language, and continuing to check specifications written in the old language would not have yielded further insight (although it might have caught more Prelude performance bugs).

Most of the specifications we wrote for Prelude express bounds on response times for various runtime operations. There were also two specifications about bounds on the percentage of operations with particular properties, capturing properties of the workload. Another specification that we tried to write (but failed because of deficiencies in the earlier PSpec model and language) expressed fairness properties of the thread scheduler.

All of the specifications were written using an earlier version of the PSpec language than the one presented in Chapter 3. The capabilities of the old language are still present in the new language, although they are now more general. Thus, the same performance bugs would have been discovered had the specifications been written in the new language. Below, the specifications are presented in the new language. The failed attempt to write the scheduler specification in the old language prompted the redesign. The scheduler specification can now be expressed, although as we will see, it is still not as elegant as it might be.

5.2.1 Response Time Specifications

One of the response time specifications, shown in Figure 5.1, expresses bounds on the durations for which interrupts are disabled on processors.

```
timed event  InterruptsOff (pid); InterruptsOn (pid);
interval  IntDisabled =
      s: InterruptsOff,
      e: InterruptsOn where e.pid = s.pid
   metrics
      time = timestamp(e) − timestamp(s),
      pid = s.pid
   end IntDisabled;
assert  {count i : IntDisabled where i.time > 75 cyc} ≤ 1
```

Figure 5.1: A response time specification for disabling of interrupts

In this specification we define the *IntDisabled* interval type. These intervals start when interrupts are turned off on a processor and end when interrupts are turned on again. The bound is expressed in the assertion, which says that interrupts are disabled for more than 75 cycles in at most one interval of that type (the long interval represents a startup transient).

The other response time specifications are all similar to the interrupts specification. They each declare event types to mark the start and end of the code path to be timed, declare an interval type that is demarcated by those event types, and assert a bound on elapsed times for all intervals of the type. The specifications we wrote were for:

- queueing messages (when necessary) during sends;
- execution on the short path for creating new messages;
- initializing some kinds of data structures;
- allocating reply codes;
- allocating object identifiers;
- performing null remote procedure calls;
- execution time of threads spawned to process incoming messages.

All of the specifications except the last one express upper bounds. The last specification (about threads created for processing messages) expresses a lower bound on the time that the threads run. If a thread does not run for

```
timed event StartSend (tid); EndSend (tid);
event MultiPacket (tid);
interval Send =
    s: StartSend,
    e: EndSend where e.tid = s.tid
  metrics
    multi = {count m : MultiPacket where m.tid = s.tid} ≠ 0
  end Send;
assert {count s : Send where s.multi} / {count s : Send} ≤ 0.05
```

Figure 5.2: A workload specification: interprocessor sends for multi-packet messages

at least some minimum length of time then it would be more efficient to do the work in-line rather than by creating a new thread.

5.2.2 Workload Specifications

One of the workload specifications, shown in Figure 5.2, expresses bounds on the percentage of inter-processor messages that require multiple packets. The percentage of multi-packet messages is a workload metric in the sense that the runtime system provides a set of services including message sending, and Prelude programs present a workload on the runtime system.

Another workload specification, expressed in implementation-level terms, concerns the percentage of messages received that get forwarded. Messages may need to be forwarded because objects in Prelude can move from one processor to another (*migrate*). The runtime system attempts to keep up-to-date information about the location of objects but may get behind if objects move too frequently. This again is a workload property in the sense that it depends upon the Prelude programs that are run (it also depends on the implementation of migration, of course). The specification defines event types corresponding to messages being received and messages being forwarded. The assertion then counts the number of forwarded messages, divides by the number of received messages to get the fraction of messages forwarded, and asserts that this fraction is less than 0.05.

5.2.3 Scheduler Specification

The Prelude runtime includes an experimental scheduler called a *priority flow scheduler*. Each thread on a processor has a *priority*, which is a non-negative number. The idea is that a thread should get an amount of processor time proportional to its priority divided by the sum of the priorities of all threads vying for the processor. Since threads get created and destroyed and can change their priorities, this priority sum is not a constant over long periods of time. The scheduler is considered to be fair if over a long enough period of time a thread gets its expected share of processor time, where the expectation is computed for each constant-priority interval. We want to write a specification asserting that threads get close to their fair shares.

The PSpec specification for the scheduler appears in Figure 5.3. Events are generated as follows. A *ChangePriority* event is generated any time the priority sum changes (e.g., when a new thread is created). After each *ChangePriority* event, a sequence of *RecPriority* events is generated, one per active thread, recording each thread's current priority (the priority for a thread is recorded as zero if the thread is not eligible to run in an interval). The timed events *SchedThread* and *DeschedThread* are generated whenever a thread gets scheduled or descheduled. Finally, *CreateThread* and *DestroyThread* events are generated whenever a thread gets created or destroyed.

We define three interval types. *TimeSlice* intervals cover the time during which some thread runs (the point at which the thread gets scheduled to the point when it is next descheduled). The elapsed times for these intervals give us the processor time that each thread receives. *SchedInterval* intervals cover the time over which the priority sum remains constant. For such intervals we compute the total priority sum, the total processor time for all threads scheduled during that interval, and a mapping from thread identifiers to thread priorities during that interval. A *ThreadStats* interval covers the lifetime of a thread. We compute metrics for the total processor time actually taken by the thread, using the *TimeSlice* subintervals, and for the processor time that the thread was expected to receive based on its priority percentage in each *SchedInterval* subinterval.

We can then assert that the actual processor time taken by a thread is close to its expected processor time. If the thread is receiving less than its fair share of the processor, then the expected time will be larger than the actual time because the thread will exist over more *SchedInterval* intervals than it should have, accumulating expected time that was never received.

```
perfspec  Scheduler;
event  ChangePriority ();
       CreateThread (tid);
       DestroyThread (tid);
       RecPriority (tid, pri);
timed event  SchedThread (tid);
             DeschedThread (tid);
interval  TimeSlice =
    s: SchedThread,
    e: DeschedThread where e.tid = s.tid
  metrics
    time = timestamp(e) − timestamp(s),
    tid = s.tid
  end TimeSlice;
interval  SchedInterval =
    s: ChangePriority, e: ChangePriority
  metrics
    totrt = {+ i : TimeSlice : i.time},
    totpri = {+ i : RecPriority : i.pri},
    threadp = {the i : RecPriority : i.tid → i.pri}
  end SchedInterval;
interval  ThreadStats =
    s: CreateThread,
    e: DestroyThread where e.tid = s.tid
  metrics
    rt = {+ i : TimeSlice where i.tid = s.tid : i.time},
    et = {+ i : SchedInterval : i.threadp(s.tid)/i.totpri × i.totrt}
  end ThreadStats;
assert  {& i : ThreadStats : i.et − 1 ms ≤ i.rt ≤ i.et + 1 ms};
end Scheduler
```

Figure 5.3: Scheduler specification

Having to generate a sequence of *RecPriority* events after each *ChangePriority* event is the "inelegance" alluded to earlier. It would be preferable simply to generate an event recording the priority of a thread when it is created or destroyed or when its priority changes; we could then use these events to demarcate constant priority intervals and to adjust the current priority sum. The problem is that in the PSpec language there is no way to compute the priority sum or the thread priority mapping for constant priority intervals defined in this way because it would require information in events occurring before the start of the interval. As we have seen before, it is not possible in the current PSpec language to compute metrics for an interval that involve events outside the interval. Recording *RecPriority* events for each constant priority interval solves the problem but is somewhat inefficient and inconvenient. The ideas presented in Chapter 6 for changes to the PSpec model would permit a more elegant scheduler specification.

5.3 Prelude Performance Bugs

As a result of checking some of the Prelude performance specifications we found four performance bugs in the Prelude runtime system. Three of the bugs were found by checking the interrupt specification, and the other was found through the assertion about time to allocate object identifiers.

We started by checking the specification about interrupts being disabled for less than 75 cycles in all but one interval, using a monitoring log generated by running a test program that used the Prelude runtime. The checker reported assertion failure. We then examined the interval dump produced by the checker to look at the intervals that lasted for more than 75 cycles. We found that some intervals lasted thousands of cycles, while others were more reasonable but still high (about 85-90 cycles).

The intervals lasting thousands of cycles were particularly mysterious so we tracked those down first. To find the cause of the problem we took advantage of the fact that the program was running on a simulator. We were able to look at the start timestamp of an overly long interval in the checker's output and determine when in the simulation that event was generated. We could then rerun the simulation, stop it during one of these overly long intervals, and examine the call stack to see what code was running at the time. The cause of the problem was obvious once we looked at the code: a thread suspended itself after having disabled interrupts without first reenabling them. This resulted from a misunderstanding on the part of the person who wrote the code about the interaction between thread suspension

and the interrupt flags; he thought that interrupts would automatically be reenabled when a thread suspended itself. In fact, interrupts finally did get reenabled through some other mechanism but the result was that they were disabled for much longer periods of time than intended.

The causes of the other long intervals with interrupts disabled were tracked down in the same way. One was due to heap allocation occurring while interrupts were off. This was simply careless coding; there was no particular reason why the allocation had to be done while interrupts were disabled. The other cause was a miscalculation on the implementors' part about how long message buffer deallocation would take. As a result of our finding, they redesigned the buffer allocation strategy for messages to make deallocation faster.

The fourth bug that we found, excessively long times for object identifier allocation, was also a problem with a buffer allocation strategy. In that case a buffer pool was too small and buffer allocation ended up reverting to memory allocation, which is too slow. The buffer pool size was increased as a result of our finding.

5.4 Evaluation

It is reasonable to ask how significant it is that we found these bugs. If the code we were testing had just been implemented the previous day, it would not have been surprising that there were performance problems. In fact, the code had been implemented several months prior to these experiments. The implementors had paid attention to performance as they were coding, and had done some tuning of the system to improve performance where it was critical. They were surprised at the discovery of these performance bugs—particularly the one that caused interrupts to be disabled for thousands of cycles. On the other hand, the pieces of the Prelude runtime that were completed had not yet seen extensive use by clients. Only simple test programs had been run. It is possible that these problems would have shown up eventually when the system was under heavier use. The problem with interrupts may not have been easy to identify, however, if the only visible effect was Prelude programs that ran more slowly than expected.

It was encouraging to see confirmation of our hypothesis that simple performance assertions are useful for identifying performance bugs. Any implementor is capable of writing the kinds of assertions we wrote—no advanced performance analysis skills are required. Although I wrote the performance assertions and did the system instrumentation, the implementors working

with me understood PSpec well enough to be able to propose specifications. Had the tools, language, and documentation been in a less experimental state, they could easily have carried out the experiments without my assistance.

It is safe to conclude that these experiments provide good preliminary support for the PSpec approach.

5.5 Methodology

Based on the experience using PSpec with Prelude and on other smaller examples it is possible to abstract a general methodology for using the PSpec tools to do performance testing and debugging. There are also some generally useful hints for writing performance specifications depending upon whether the intended use of a specification is regression testing, continuous monitoring, or performance debugging.

5.5.1 Process

The first step in using the PSpec tools is to decide what performance metrics are relevant for a program, and what assertions can be made about those metrics. For guidance in this step, the discussion in Chapter 2 might be helpful in providing ideas about useful metrics, or the many books that have been written about performance evaluation and measurement might help (e.g., [10, 11, 18]). Obvious metrics to consider are response time, throughput, resource utilization, and workload properties (particularly where assumptions have been made in the implementation). Also, analysis, simulation, and measurement tools that provide insight into the performance of a program and help a designer or implementor to develop performance expectations can be employed here if they are available.

The next step is to express the performance metrics and assertions in the PSpec language. This involves declaring events and intervals that are needed to define the metrics, and then writing assertions using those declarations. Unknown constants can be included as symbols. The specification writer should have a rough idea of how the events can be logged from the program, but need not actually add the logging code during this step.

These first two steps can (and probably should) be performed even before a system is implemented. It is well-known folk wisdom that designing performance into a program from the start produces better results than trying to add it in after the fact. Writing performance specifications can help

to focus and clarify a designer's or implementor's performance expectations. It is particularly useful for implementors to capture any assumptions they make about the workload on a system or about resource availability. These are the types of decisions that are easily forgotten later on if they are not documented, and their failure to hold is frequently a cause of performance bugs. Of course, these steps can also be performed at any later stage in the development cycle.

The next step is to annotate the program code to generate monitoring logs. This step, obviously, must be performed during or after the implementation. If the program is to be simulated before the actual implementation is done then the simulator can also be instrumented to produce monitoring logs for the checker. The simulated version is just a different implementation of the program. The specifications could be checked first for the simulated version and then later for the real version. If the specification is to be used for regression testing, the driver programs must also be written and annotated as necessary.

Before a specification can be checked, values for any unknown constants must be determined. This can be done using the solver, or any other available method. As described in Chapter 4, using the solver generally involves running the program under conditions that produce monitoring logs suitable for estimating the unknowns.

Once the constants are filled in, the program can be run to generate monitoring logs to be input to the checker along with the specification. The setup for this will depend upon whether the specification is intended for regression testing, for continuous monitoring, or for debugging, and the details will be specific to the particular system being used. For Proteus, it is merely a question of whether a specification is checked after every simulation run (in effect, continuous monitoring) or only after designated runs (regression testing or debugging). In non-simulated systems, the setup will probably be more complicated.

The final step is to track down the causes of any assertion failures reported by the checker. Again, how this is done depends upon the facilities available in the particular system. For the simulator, it is generally a matter of examining the output of the checker and rerunning the simulation, stopping it at points to examine the state. The process will probably be more difficult in non-simulated systems, particularly when the executions are not easily reproduced. Tracking down the cause of assertion failures may also involve writing more specifications and going through the whole process again. Like program development, using performance assertion checking to debug

and test the performance of programs is a cycle rather than a straight-line process.

5.5.2 Hints for Writing Performance Specifications

During the first two steps described above—deciding upon assertions and metrics and translating them into PSpec—it is worthwhile to think about the uses to which the specifications will be put. The intended use affects the amount of logging overhead that can be tolerated and the assumptions that can be made about workloads.

For regression testing, a high monitoring overhead is more easily tolerated, both because the tests will not be run that often, and because they will not be run as part of normal operation. Therefore, more and lower-level details can be gathered about performance. For example, our interrupt specification is probably one we would want to check during regression testing and not during continuous operation.

Also, during regression testing we have a fair degree of control over the workload on the system. We can use this knowledge when writing assertions; sometimes it may be easier to write assertions knowing that the system will have a particular workload, while it is not easy to write an assertion that detects that workload.

For specifications that are checked continuously during normal system operation, we will probably have to ensure that the overhead of generating logs is reasonably low. This means that the assertions will have to be less detailed than is possible when doing regression testing. For example, we might be able to check the specification about the percentage of send operations that are multi-packet, or about cache hit ratios in a file system. If those specifications as written still result in too high an overhead for logging, we could perhaps change them so that fewer events are logged to give approximately the same data. For example, in the case of sends, rather than logging an event for each send we could keep counters in the program that counted all sends and all multi-packet sends. The value of these counters could be output to the log periodically. This would leave us with less information if we then had to track down the cause of an assertion failure, but the tradeoff might be worthwhile.

During continuous monitoring, we also have less control over the workload on the system. This means that the assertions will have to apply over a broad range of workloads, either by asserting performance properties that hold for all workloads, or by identifying applicable workloads in the asser-

tions. It is also particularly useful to check implementation assumptions about workloads during continuous monitoring.

Performance debugging involves tracking down the cause of a known performance problem. During debugging we will probably be writing assertions for various levels of abstraction in a system, going to lower levels and more detail as we trace a problem. Repeatability helps immensely here since it is usually impossible to predict before running a program and examining its monitoring output all of the information that will be needed to locate a bug. A simulator is a particularly good environment for tracking down performance bugs because executions are repeatable and it is possible to stop the program to examine what is going on without affecting the outcome of the execution. Performance debugging in non-simulated systems is much more difficult, and we do not yet have any experience using PSpec for debugging in that situation.

Chapter 6

Extensions and Future Directions

This chapter is a speculative discussion about future work. The first two sections deal with what could be called "immediate future work," while the third section discusses a longer-term direction. The first section covers issues related to using the PSpec language and tools in non-simulated systems. The discussion contains many more questions than answers because of a lack of experience using the tools for such systems. The second section presents ideas for how the language could be changed or extended to solve some of the expressiveness problems we encountered in earlier chapters. None of these ideas have been worked out to the point where they are ready to be incorporated into the language. The third section discusses the use of performance specifications as interface documentation.

6.1 "Real" Systems

The PSpec language and tools were intended from the start to be useful in "real" (non-simulated) systems. The Proteus simulator certainly is real in the sense that people use it for their real work, and not just for toy problems; the programs developed on the simulator are also executed on parallel machines. However, the simulator environment provides certain amenities that are not available in non-simulated systems, such as the ability to replay executions and to do zero-cost monitoring. Since performance specifications ought to be checked throughout the lifetime of a program, it is important that they can be checked in production environments as well as on simulators. Also, we will want to employ the tools in environments for which no simulator exists.

One of the first questions is: will the monitoring overhead be tolerable? Overhead includes both the time to generate the log data, and the space

to store it. There is good reason to believe that monitoring overhead will not be the downfall of this approach. One way to bring monitoring overhead under control is to adjust the level of detail at which specifications are written. For the purposes of continuous monitoring, specifications can be written to detect gross performance problems, and to monitor gross characteristics of workloads. Regression tests could use more detail because the higher monitoring overhead would be more tolerable (except when the monitoring actually changes the behavior of the system). Also, as mentioned in the previous chapter, by keeping small amounts of additional state in the program being monitored, in the form of counters and timers, one can reduce the number and frequency of events written to a log.

Being able to discard log data once the assertions pertaining to it have been checked should help with space problems. This is one of the advantages of having performance specifications—they help to identify the interesting data corresponding to the failure of the system to meet expectations. In the end, it will be a question of whether the benefit of performance assertion checking justifies the cost of monitoring. Monitoring will always have some cost, but if performance problems can be detected and corrected early, the extra cost can be justified. Without monitoring, we are once again without a solution to the problem of finding performance bugs.

To make the PSpec tools work in real systems, several questions relating to continuous monitoring must be addressed. What is the mechanism for setting it up? What do aggregate expressions in specifications mean in that case? What happens when assertions fail? How do we figure out what information can be discarded and what should be saved? Must we restrict specifications to use a subset of the PSpec language that can be checked in a single pass through a log? Perhaps what we really want is "periodic" rather than "continuous" monitoring, where logging is turned on and specifications are checked periodically rather than all the time.

There are several possibilities for handling aggregate expressions during continuous monitoring. First we have to decide what the meaning of an aggregate expression should be. For example, what does it mean to write an assertion about an average that is checked continuously? Probably, what we want is averages over particular periods of time—maybe ten-second averages or ten-minute averages or day-long averages. The current PSpec language definition says that a top-level aggregate expression ranges over the entire log against which the specification is checked. One obvious solution is to break up logs into lengths covering whatever period of time is desired and feed those to the checker. Another possibility is to write explicit assertions

that apply over the desired fixed-length periods. We considered such specifications in Chapter 2 where we looked at assertions about ten-second average throughputs. As we saw, such specifications are not difficult to write, providing time markers can be generated in logs.

The questions of what to do when assertions fail and what information can be discarded during continuous monitoring remain to be answered. Probably for assertion failures we will want some mechanism for logging the failure so that a person will eventually see it, or to otherwise bring it to someone's attention. When assertions fail we probably want to save more data than just the data for which the assertions failed. We might like to save portions of a log for periods before and after the failure to provide context for tracking down its cause. In order to know the right answer here we need more experience using the tools in these situations.

The question of how to track down causes of assertion failures is also an issue for regression testing and, of course, for performance debugging. Debugging a program interactively, as we did to find the Prelude performance bugs, is helpful if it is possible. Depending upon the system for which the specification is being checked, this may not be possible to the same extent that it was with the simulator. Again, more experience is required to know what tools and techniques will work.

Finally, in real systems, logs may be generated in several different pieces that need to be merged before they are given to the checker. Work on monitoring in distributed and parallel systems has addressed this problem. One solution involves using timestamps and synchronized clocks. Another solution is based on Lamport's "logical clock" concept [21]. Other system-specific solutions are also possible (e.g., see [31]).

Doubtless there are other problems that will have to be solved to make the PSpec tools useful in real systems. Experience will tell.

6.2 Language Extensions

We now turn to ideas for extensions to the PSpec model and language that could help solve some of the problems with expressive power that we encountered earlier.

6.2.1 Computing State From a Log

In Chapter 2 we saw an example where we wanted to compute a boolean metric for a Read interval that would be true if there were any concurrent

Write intervals. We had difficulty because, while it is easy to detect whether any Write intervals start or end during a Read interval, it is not easy to detect whether there are any Write intervals that start before the Read and are still open during the Read. The problem, in essence, is that a metric expression in an interval declaration cannot refer to events outside the interval (and in particular, before the interval). This prevents us from computing a metric whose value is the state of the computation at some point in an interval using events from the beginning of the log up to that point. We saw another example where we wanted to compute state in Chapter 5 with the Prelude scheduler; there we wanted to compute the current priority sum and the mapping of threads to priorities in constant priority intervals. This problem of not being able to compute state is perhaps the most serious defect in the PSpec model and language.

One obvious solution is to change the model so that metric values may be computed from events preceding an interval as well as within an interval. The question would then be how to reflect this in the language. There are two possible solutions that I have considered, neither of which is entirely acceptable. It is worth recording them here because one or the other may be a step towards a good solution. The solution that seems most promising involves modifying the definition of aggregate expressions to allow aggregates in a metric definition to range from the beginning of the log to some point in the interval containing the metric. The less attractive solution involves adding a notion of "state variable" to the language. I will describe each of these and show how they could be used to express the scheduler specification. (The original scheduler specification appears on page 77.)

First let us consider the solution involving aggregate expressions. In the current design, the range of an aggregate expression is implicit from its context; aggregate expressions appearing at top-level (in assertions, def statements, or solve declarations) range over the entire log, while aggregate expressions in metric definitions range over the events and sub-intervals of the interval for which the metric is defined. The proposal is to add a third possibility, which is an aggregate expression that ranges from the beginning of the log up to some point in the log (through some event); it is not clear how much flexibility should be allowed in specifying the termination point for the aggregate.

Suppose the scheduler specification is modified so that instead of *ChangePriority* and *RecPriority* events, we have an event type:

event *ChangeThreadPri (tid, newpri).*

An event of this type is generated every time a thread is created or destroyed, or when its priority changes, and records the thread identifier and the thread's new priority. (When a thread is destroyed its new priority is recorded as zero.)

We then define *SchedInterval* intervals to start and end with events of type *ChangeThreadPri*. To compute the *threadp* mapping metric for these intervals, we could write one of our new aggregate expressions. Assuming that we can somehow indicate that the following aggregate expression ranges from the beginning of the log through the *ChangeThreadPri* event that starts an interval, we could write:

$$threadp = \{last\ c : ChangeThreadPri : c.tid \rightarrow c.newpri\}.$$

This expression constructs a mapping from thread identifiers to the last priority value recorded for that thread identifier. The priority sum for an interval of type *SchedInterval* can then be computed by summing the priorities in the range of the *threadp* mapping.

The problem here is how to specify the range of the new kinds of aggregate expressions. Aggregate expressions are concise now because the range is implicit, but it would probably have to be made explicit in the new proposal. Also, in this example we needed the aggregate to range through the first event of the interval, but in other examples, we might want the aggregate to range up to but not including the first event, or even up to some other event contained in the interval. It is not clear how this could be expressed cleanly. The appeal of this approach is that the language remains assignment-free and it uses an existing feature of the language with some modifications rather than introducing an entirely new concept.

The second proposal for computing state in specifications is to add state variables. We (logically) augment a log by adding a *state* after each event. A state is a mapping from state variable names to values. A variable name can be mapped to different values in different states. A specification writer introduces a state variable by giving its initial value and specifying how its value at any point in the log is altered by each kind of event. There also must be a way to refer to the value of a state variable before or after any given event.

For the scheduler example, we could introduce a state variable called *curPriorities* that holds the mapping from thread identifiers to thread priorities. The mapping is initially empty. A *ChangeThreadPri* event c either replaces the priority value mapped to $c.tid$ if there is one, or adds $c.tid$ to the domain of *curPriorities*, mapping it to $c.newpri$. The value of the

threadp metric for a *SchedInterval* interval is then the value of *curPriorities* in the state following the start event of the interval. The specification might look something like the following:

```
event  ChangeThreadPri (tid, newpri);
       ⋮
var  curPriorities
  init empty,
  mod c: ChangeThreadPri ⇒ curPriorities(c.tid) := c.newpri;
       ⋮
interval  SchedInterval =
    s: ChangeThreadPri, e: ChangeThreadPri
  metrics
    threadp = s^curPriorities
    ⋮
  end SchedInterval;
       ⋮
```

The notation $s\hat{}v$ is intended to mean the value of state variable v following event s. The var statement declares the state variable *curPriorities*, gives its initial value, and specifies how the value is modified by *ChangeThreadPri* events. Implicitly, the state variable's value is not changed by any other types of events.

State variables might work, but they seem less appealing than a solution that extends the notion of aggregates. One of the reasons is that specifications become harder to read; to understand what the value of a state variable will be at any point the reader has to reason about how the value can change throughout the log up to that point. Also, state variables provide some capabilities already provided by aggregate expressions. It seems unnecessary to have both mechanisms in the language. We could consider removing aggregates and using only state variables, perhaps allowing state variables that are local to intervals for computing metrics. However, in my opinion, this makes specifications even less readable because more operational reasoning is required.

6.2.2 Virtual Events

Another idea for an extension to the PSpec language is to add a notion of *virtual events*. A virtual event would be like a real log event, sequenced

with real events, and having named attributes. The difference is that virtual events would be fabricated by the checker based on a specification and would not actually be contained in a log. Having virtual events in the language might save us from having to log extra events from a monitored program, or from having to preprocess a log to add events before giving it to the checker.

We have already seen several examples in previous chapters where virtual events could be useful. In one example we needed events that marked off fixed intervals of time in a log. If all events in a log have timestamps, such interval markers could be added in a post-processing phase rather than having to be generated while the monitored program is running. Providing a way to define these as virtual events would allow the post-processing to happen along with specification checking.

Another example where virtual events would have helped was mentioned in Chapter 3, where I discussed defining an interval that starts when a file is created and ends when it is either deleted or overwritten. The problem is that we need to identify the end event for an interval type using a disjunction of event types, where the language allows only a single event type. We could have generated additional events during monitoring that corresponded to the case of "either deleted or overwritten" but this information is already present in the log so it seems wasteful to generate extra events at the time when the overhead is least tolerable. We also could have post-processed the log to add these events. The ability to define virtual events would eliminate the need for post-processing. Instead, we could imagine defining a virtual event corresponding to the "deleted or overwritten" case that is generated after each delete or overwrite event. We could then use the virtual event type to identify intervals.

Virtual events could also be useful for turning intervals into events, in effect. We could imagine defining a virtual event type corresponding to intervals of a specified type. Whenever an interval of the type occurred in the log (at the point where its end event occurred) the virtual event would occur. Then these virtual events could be used to demarcate other intervals. This gives us the capability of defining intervals that are demarcated by intervals rather than just events. We have not yet run across any examples where this capability is required, but it might be useful.

6.3 Performance Specifications as Interface Documentation

Ideally, performance specifications ought to serve an analogous role to functional specifications. A functional specification abstracts from a computa-

tion by describing its intended effect, namely, the input-to-output relation that it computes. Any number of different implementations may satisfy the same functional specification, computing the input-output relation in different ways. A functional specification hides the details of an implementation that are deemed irrelevant for proving the correctness of a program that uses the implementation. In systems without hard real-time constraints, program correctness is not usually tied to performance, and so functional specifications often do not include information about performance.

A performance specification ought to be a document that accompanies a functional specification, and similarly, abstracts from the details of an implementation. It should describe the values of interesting performance metrics that the client of the implementation can expect to see under various operating conditions. Many different performance specifications could accompany the same functional specification; a performance specification could be viewed as a further constraint on implementations.

Such performance specifications could help in building programs that perform well. In order to choose implementation strategies that result in good program performance, a programmer must know what performance to expect from the subsystems used by the program, and how the performance of the various subsystems will affect the performance of the program as a whole. Performance specifications could help with the former problem by documenting the expected performance of subsystems. They would not, however, help the programmer to determine what operating conditions will prevail when a subsystem is used (information that will be needed in order to interpret the specification), nor how to determine the resulting performance of a program when many subsystems are used together.

Once a program is implemented, interface-level performance specifications could aid in assigning blame during performance debugging. A specification is a contract for a service to be provided by a subsystem to its client. If a subsystem fails to meet its specification, either the subsystem has a bug or the specification is wrong. In either case, the implementor of the client is justified in complaining to the subsystem's implementors. If a program does not perform as well as the programmer expects but the subsystem suspected of causing the problem is actually meeting its specification, there are a number of possibilities. It could be that conditions in the system are different than the programmer expected, and so the subsystem is operating in a different performance range than expected. Some other subsystem may be failing to meet its specification and so may be at fault. Another possibility is that all subsystems are meeting their specifications and the programmer's

performance expectations are simply unreasonable.

The significant difference between the interface-level performance specifications that I have in mind and the PSpec performance specifications that we saw earlier is that interface-level specifications are contracts. They would allow programmers to debug the performance of application programs without understanding the implementations of all the subsystems used by the program. As soon as it is determined that a subsystem failed to meet its performance specification, the programmer would know who to complain to and the responsibility then belongs to the implementor of the subsystem.

There are (at least) several reasons why writing interface-level performance specifications is difficult. One was mentioned in Chapter 2: it is not always easy to characterize performance in interface-level terms. Performance of a subsystem may depend upon the history of its use, which is not easy to describe in terms of the requests at its interface. This was the reason that we wrote elapsed time assertions for file system Read operations in terms of a metric that indicated whether there was a cache hit, rather than characterizing the performance of a Read operation that was preceded by a particular pattern of file system requests. A solution for this problem may be to introduce concepts like "cache" at the interface level for the purposes of writing performance specifications for clients. In this case though, it would be difficult to describe in precise terms to the client how to use the file system to obtain good cache behavior. A related problem is that the performance of a subsystem may depend upon lower-level subsystems that it uses. It is hard to see how to include this information in an interface-level specification.

Another reason why interface-level performance specifications are difficult to write is that it is not as easy to get modularity with performance as it is with functionality. *If* a performance specification can identify all the possibly important input performance variables for a subsystem, then an implementation change in another subsystem will not invalidate the specification. The "if" is emphasized because this condition can be difficult to satisfy for two reasons. First, identifying all the variables that affect performance may be difficult because some of them have only small effects or are masked entirely by other performance variables in a given implementation. If there is no way to detect these hidden performance variables, a change in one subsystem could invalidate the specification for another by causing a previously unimportant input variable to become important. Second, it is probably not a good idea to try to express the performance contribution of all input variables with small performance effects in a specification; the

specification is intended to serve as documentation for users and as input to a checker and cluttering the specification with relatively unimportant details may obscure more important information.

A third factor working against interface-level specifications is that the performance of a subsystem depends upon the entire workload presented by all its clients. In general, it is not possible to characterize the performance that one client will see based on that client's workload alone. Even knowing the entire workload, it may only be feasible to specify characteristics of the performance delivered to all the clients and not specifically to one client. Consequently, to make use of a specification, clients of a subsystem will need information about other clients that use the subsystem. This is a failure of modularity in another form. All hope is not lost however, since a monitoring system that gathers information to check whether a subsystem is meeting its performance specification could also make the information about the entire workload available to each client.

Despite these difficulties, the idea of interface-level performance specifications is worth pursuing because the payoff could be large. It is possible that the PSpec language, perhaps with some modifications, could provide a fine vehicle for expressing such specifications. The first problem though is to figure out what the content of the specifications should be, and then we can consider the notation.

Chapter 7

Conclusion

Never promise more than you can perform.
—Publilius Syrus, Maxim 528

We have now seen all of the components of the PSpec approach along with examples and evidence of how it can be used for performance debugging and testing. This chapter concludes by summarizing the contributions of the research and mentioning the directions for future research that are likely to be most critical to the success of performance assertion checking in the long run.

7.1 Summary of Contributions

The main idea underlying this research is that explicit and precise performance assertions can help to automate performance regression testing and continuous monitoring of systems, and thereby help to find performance bugs. The PSpec language and tools are a realization of this idea.

The PSpec language is a notation for describing predicates on monitoring logs. Its design is geared towards expressing performance assertions, but in fact, the language is more general than that. For example, it can also express (to a limited extent) well-formedness properties on logs. A monitoring log provides a simple interface between the assertion language and the program whose performance is being described. It abstracts away the idiosyncrasies of the system being monitored to capture those facts about executions that are relevant for performance assertions. The existence of this interface permits a single implementation of a set of tools for processing performance assertions; the tools can be used with monitoring logs from a wide range of systems.

The PSpec checker and solver are two tools that are geared specifically towards writing and checking performance assertions. As discussed in Chapter 4, other tools could also be designed to fit into the performance assertion framework and augment the capabilities of the checker and solver. For this research, not much attention has been given to integrating the PSpec tools with existing performance tools but that would be a useful next step.

One advantage of the PSpec approach is that it has a low startup cost. Performance assertions can be written for any piece of a system that can be monitored, at any level of abstraction. There is no requirement that performance specifications be complete, or that they be provided for all modules in a system, or that they be written in interface-level terms for a module. An implementor can write whatever assertions may help to locate problems, incrementally adding more assertions and more monitoring as necessary.

PSpec provides a systematic approach to performance testing that can replace current *ad hoc* approaches in many situations. Implementors have always had the ability to generate monitoring logs and process them to check properties of executions, but they seldom do this because it is time-consuming to write the log processing programs and to set up the monitoring. Also, when such special-purpose instrumentation is produced for the performance testing phase of program development, it tends to become obsolete over time. Part of the idea behind PSpec is that the performance testing phase should continue throughout a system's lifetime; providing a general-purpose set of tools that can be maintained and understood by everyone can help to make this happen.

I began with the hypothesis that relatively simple performance assertions would be useful for finding performance bugs. One of the reasons that so many systems perform badly is that no attempt is made to study performance, or only initial performance studies are conducted just prior to releasing a system and no followup studies are done to track changes in performance over time. Tools that take advantage of the knowledge that programmers already have and that facilitate and encourage simple performance studies that continue throughout the lifetime of a system would thus be valuable. Preliminary experience with PSpec has borne out the hypothesis; the assertions that we used to find the Prelude performance bugs were simple and obvious. This observation, along with the fact that the PSpec approach is more structured and general than *ad hoc* approaches, bodes well for the prospect of programmers actually using the PSpec tools and thereby producing systems with better performance than we see today.

7.2 Directions for Future Research

If performance assertion checking is to become a ubiquitous component of the programmer's toolkit, additional issues beyond the scope of this dissertation must be addressed. Chapter 6 discussed some questions related to making the PSpec tools work for non-simulated systems. Some additional questions are:

- How do heterogeneity and portability affect performance specifications? How do we write checkable specifications for distributed systems with components that have different performance characteristics (particularly where the software masks the differences from a functional standpoint)? Can we parameterize specifications so that minimal changes are required when a system is ported to a new architecture or configuration? (The solver should help here.)

- How does fault-tolerance affect the kinds of performance assertions we write? Because fault-tolerant systems try to provide graceful degradation, component failures often result in performance degradation rather than system failure. What kinds of performance assertions will be useful for such systems?

- The presentation here has focused on writing performance assertions for single runs of programs, but in fact, the same techniques can be used to write assertions about multiple runs. Are any changes required to the PSpec language or tools to support the checking of multiple program runs?

- A common cause of poor performance is high contention for base system resources such as memory pages, processor time, or network bandwidth. When such resources are in high demand, performance tends to degrade for the entire system, rather than just for individual programs. How should we write performance specifications to identify performance problems due to base resource contention? Can we build tools that help to automate the production of such specifications?

- How can we make the solver more helpful?

All of these questions are fruitful directions for future research.

References

[1] William Alexander and Richard Brice. Performance modeling in the design process. In *AFIPS National Computer Conference*, volume 51, pages 257–262. AFIPS, 1982.

[2] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '90*, May 1990.

[3] Ziya Aral and Ilya Gertner. Non-intrusive and interactive profiling in Parasight. In *Proceedings of the ACM PPEALS Conference*, pages 21–30. ACM, 1988.

[4] Hagit Attiya and Nancy A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. Technical Memo MIT/LCS/TM-403, Massachusetts Institute of Technology, Cambridge, MA 02199, July 1989.

[5] David Bernstein, Anthony Bolmarcich, and Kimming So. Performance visualization of parallel programs on a shared memory multiprocessor system. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 1–10, August 1989.

[6] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[7] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high-performance parallel architecture simulator. In *Proceedings of the*

1992 ACM SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems, pages 247–248, Newport, RI, June 1-5 1992.

[8] K. M. Chandy, J. Misra, R. Berry, and D. Neuse. The use of performance models in systematic design. In *Proceedings of the AFIPS NCC*, volume 51, pages 251–256. AFIPS, 1982.

[9] Digital Equipment Corporation. *pixie(1)*. Ultrix 4.0 General Information, Vol. 3B (Commands(1): M-Z).

[10] Domenico Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.

[11] Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.

[12] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, pages 163–173. ACM, May 1988.

[13] Yogesh Gaur, Vincent A. Guarna Jr., and David Jablonowski. An environment for performance experimentation on multiprocessors. In *Proceedings of SUPERCOMPUTING '89*, pages 589–594. IEEE Computer Society and ACM SIGARCH, November 1989.

[14] S. L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: A call graph execution profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, June 1982.

[15] Karen J. Hay, Sanjay Manchanda, and Richard D. Schlichting. Proving real-time properties of distributed programs. Technical Report TR 88-40b, The University of Arizona, Tuscon, Arizona 85721, December 1988.

[16] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[17] Robert V. Hogg and Elliot A. Tanis. *Probability and Statistical Inference*. Macmillan Publishing Company, New York, 3rd edition, 1988.

[18] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley and Sons, Inc, 1991.

[19] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[20] Teemu Kerola and Herb Schwetman. Monit: A performance monitoring tool for parallel and pseudo-parallel programs. In *Proceedings of SIGMETRICS 1987*, pages 163–174. ACM, 1987.

[21] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[22] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.L. Popek. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2), February 1977.

[23] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1984.

[24] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the Fifth DCS*, pages 515–522. IEEE, May 1985.

[25] Ted Lehr, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing performance debugging. *IEEE Computer*, 22(10):38–51, October 1989.

[26] Nancy A. Lynch and Hagit Attiya. Using mappings to prove timing properties. Technical Memo MIT/LCS/TM-412, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1989.

[27] Allen D. Malony, David H. Hammerslag, and David J. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):19–28, September 1991.

[28] Barton P. Miller, Morgan Clark, Steven Kierstead, and Sekl-See Lim. IPS-2: The second generation of a parallel program measurement system. Computer Sciences Technical Report 783, University of Wisconsin—Madison, Madison, Wisconsin, August 1988.

[29] Greg Nelson, editor. *Systems Programming With Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[30] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24. ACM, December 1985.

[31] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 183–197. ACM, October 1991.

[32] Zary Segall and Larry Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, pages 22–37, November 1985.

[33] Sanjay Sharma, Allen Malony, Michael Berry, and Priyamvada Sinvhal-Sharma. Run-time monitoring and performance visualization of concurrent programs for shared memory multiprocessors. Technical Report CSRD 987, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2932, March 1990.

[34] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[35] Mary Shaw. A formal system for specifying and verifying program performance. Technical Report CMU-CS-79-129, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1979.

[36] Connie Smith and J.C. Browne. Aspects of software design analysis: Concurrency and blocking. In *Proceedings of Performance '80*, pages 245–253. ACM, 1980. Also Performance Evaluation Review 9(2). 7th IFIP W.G. 7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation.

[37] C.U. Smith and J.C. Browne. Performance engineering of software systems: A case study. In *Proceedings of AFIPS NCC*, volume 51, 1982.

[38] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[39] Murray R. Spiegel. *Theory and Problems of Statistics*. Schaum's Outline Series in Mathematics. McGraw-Hill Book Company, 2nd edition, 1988.

[40] Ben Wegbreit. Verifying program performance. *Journal of the ACM*, 23(4):691–699, October 1976.

[41] W.E. Weihl, E.A. Brewer, A. Colbrook, C.N. Dellarocas, W.C. Hsieh, A.D. Joseph, C. Waldspurger, and P. Wang. PRELUDE: A system for building portable parallel software. Technical Report MIT/LCS/TR-519, MIT Laboratory for Computer Science, October 1991.

[42] W.E. Weihl, E.A. Brewer, A. Colbrook, C.N. Dellarocas, W.C. Hsieh, A.D. Joseph, C. Waldspurger, and P. Wang. PRELUDE: A system for portable parallel software. PARLE '92 Parallel Architectures and Languages Europe, June 15-18 1992.

[43] Dieter Wybranietz and Dieter Haban. Monitoring and performance measuring distributed systems during operation. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 197–206. ACM, May 1988.

Appendix A

PSpec Language Reference Manual

A.1 Introduction and Definitions

PSpec is a language for writing performance specifications. We assume that there is some facility for generating monitoring logs from programs for which we want to write specifications. A specification describes a set of assertions (a predicate) expected to hold for monitoring logs of interest.

A *monitoring log* is a sequence of primitive components called events. Each event has a type and a sequence of named, numeric-valued attributes. An event type in the PSpec language corresponds to an event type in a monitoring log. The mechanism for establishing this correspondence is implementation-dependent.

An *interval* corresponds to a subsequence of a log starting at some start event, ending at some end event, and including all events in the log between the start and end events. An interval has associated metrics, which are named and have values (not necessarily numeric). Intervals may be disjoint, may overlap, or may nest. An interval i is nested inside another interval j if i's start and end events are between j's start and end events in the log.

Values are mathematical entities with types. Some examples of values are events, intervals, numbers, and booleans.

An *expression* specifies a computation that produces a value.

An *identifier* is a symbol declared as a name for a value. The region of a specification over which a declaration applies is called the scope of the declaration. The outermost or top-level scope of a specification is called the global scope. Event and interval type names and declared constants are all in the global scope. In addition, interval declarations and aggregate expressions (to be described later) introduce local scopes. Scopes nest, with names in the global scope accessible from all enclosed scopes (though there

are certain restrictions, described later). Identifiers must be declared before they are used in expressions.

A.2 Types

Every value in a specification has a type that dictates how the value may be interpreted. With the exception of event and interval types, types are never named explicitly in specifications. However, the type of any value or expression can always be inferred statically.

The base types are number and boolean. From these base types, triple and mapping types can be constructed.

A triple consists of three numbers and represents a measurement with associated error (or it may represent a combination of measurements). The triple $[t, p, m]$ represents a number in the range $[t - m, t + p]$, where t is the measured or "favored" value. p and m are always non-negative.

A mapping is a partial function from integers to values.

In addition to the above types, specification writers can declare event and interval types. The declarations for these types are described in the next section. Event and interval types have names that uniquely identify the types (i.e., two types are the same if they have the same name).

A.3 Declarations

A declaration introduces a name for a constant, event type, or interval type into the global scope. The name is available in all expressions that follow the declaration. An interval type name is not available in metric definitions for the interval type (thus, interval declarations are not recursive). It is an error to redeclare a name in the same scope (but a name may be redeclared in a nested scope).

A.3.1 Constants

If *id* is an identifier and e is an expression, then:

def *id* $= e$

declares *id* as a constant bound to the value of e. e is evaluated in the global scope. The declaration ***def*** *id* $=$ *?* declares *id* to be an unknown. Unknowns are recognized by the solver.

A.3.2 Event Types

If *id* is an identifier and *alist* is a comma-separated list of identifiers, then:

event *id (alist)*

and

timed event *id (alist)*

declare *id* as an event type. Events of the type have attributes named in *alist*. The second form of declaration also indicates that events of type *id* have an implicit timestamp attribute, which is used by the *elapsed* function on intervals and can be accessed by the *timestamp* function on events.

A.3.3 Interval Types

Interval type declarations serve two purposes: they introduce new interval types and they also identify the set of intervals in a log. An interval declaration provides predicates for determining whether an event in the log is the start or end event for an interval of the type, and expressions for computing the metric values for an interval of the type.

If *id*, *s*, and *e* are identifiers, *stype* and *etype* are event type names, and *spred* and *epred* are boolean-valued expressions, then:

interval *id* =
    s: stype ***where*** *spred,*
    e: etype ***where*** *epred*
  metrics
    mlist
  end *id*

declares an interval with type name *id*. *mlist* is a comma-separated list of metric definitions of the form

m = *expr*

where *m* is an identifier and *expr* is an expression.

The declaration defines an interval of type *id* to be one that has a start event *s* of type *stype* for which *spred* is true, and an end event *e* that is the next event in the log of type *etype* following the start event for which *epred* is true.

The definition of intervals may be modified slightly by inserting the keyword ***nested*** before ***interval***. A nested interval type has the further condition that the end event e is not also an end event for any other interval of type *id* that starts after s. Thus, without the nested restriction, multiple intervals of the type may share the same end event. With the restriction, all intervals of the type will have different end events. It is always the case that multiple intervals of different types may share end events.

For each interval of type *id* in the log, its metrics are computed as follows. A new scope is introduced with s and e bound to the start and end events for the interval. The metric expressions are evaluated in this scope and their values are bound to the metric names for the interval. Metric definitions are, in effect, simultaneous—they may not reference each other. Also, because of an implementation efficiency consideration, metric expressions that aggregate over events or intervals may not refer to e (the end event).

The ***where*** clause for the start or end event may be omitted, in which case it defaults to *true*. The ***metrics*** *mlist* clause may be omitted, in which case the interval has no metrics (but the *elapsed* function may still be applicable for the interval).

The identifier s is available in *spred* and *epred*. The identifier e is available in *epred*. Both identifiers are available in the metric expressions, as explained above. The identifier *id* cannot be referenced inside *mlist*. There is also the restriction that expressions in *mlist* cannot reference identifiers that are defined to have values computed using aggregate expressions.

A.4 Assertions

An assertion is a predicate (boolean-valued expression) that is expected to be true when a specification is checked against a log. If e is a predicate, then

assert e

is an assertion that e should evaluate to *true*. e is evaluated in the global scope.

A.5 Solve Declarations

A tool called the *solver* accepts specifications with constants declared as unknowns and estimates values for the constants. Solve declarations provide

guidance to the solver.

If *id* is an identifier, *idtype* is an event or interval type, *v* and *c* are constants declared as unknowns, *pred* and *e* are expressions, and *m* is a mapping-valued expression then:

solve *e*

and

solve ***data*** *id : idtype* ***where*** *pred : e,* ***var*** *v,* ***cor*** *c*

and

solve ***data*** *id* ***in*** *domain(m)* ***where*** *pred : e,* ***var*** *v,* ***cor*** *c*

are solve declarations. See the solver documentation for more information about how solve declarations are used.

A.6 Specifications

If *id* is an identifier and *stmts* is a semi-colon separated list of declarations, assertions, and solve statements, then:

perfspec *id*
stmts
end *id*

is a specification. *id* must be different from any top-level identifier in *stmts*.

A.7 Expressions

An expression specifies a computation that produces a value. Expressions are either operands (identifiers or literals), operators applied to arguments that are themselves expressions, triple constructors, or aggregate expressions.

The operators that have special syntax are classified and listed in decreasing precedence order in Figure A.1. All infix operators are left associative. Parentheses can be used to override precedence rules.

f(x)	function or mapping application
i.f, *n us*,*n ms*, etc.	infix dot for field access, numeric literals with time units
−	unary minus
∗ / *div mod*	infix arithmetics
+ −	infix arithmetics
= != < <= >= >	infix relations
!	prefix "not"
&	infix "and"
|	infix "or"
=>	infix "implies"
−>	infix mapping constructor

Figure A.1: Operator precedence

A.7.1 Literals

There are two kinds of literals: numeric and boolean. The boolean literals are *true* and *false*. Numeric literals denote non-negative numbers and use the Modula-3 syntax for integer, real, and longreal literals [29]. All numbers are converted into longreal internally, but we can still check when necessary whether a number has an integral value.

A.7.2 Triple Constructors

If e, p, and m are numeric-valued expressions, then $[e, p, m]$ is the triple whose components are the values of e, p, and m, in that order. Such a triple represents the range $[e - m, e + p]$ with e being the measured or "preferred" value. Both p and m must be non-negative.

A.7.3 Mappings

If i is an integral number-valued expression and v is any expression then i −> v is the single-element mapping with the value of i mapped to the value of v.

If m is a mapping and i is an integral number-valued expression, then *m(i)* evaluates to the value to which m maps i. i must be in m's domain.

The expression *mapped(m, i)* evaluates to *true* if and only if i is in m's domain.

Some of the arithmetic and logical operators are overloaded to work on mappings. These operators provide various ways of combining mappings. In particular, +, ∗, &, |, *min*, and *max* can take mappings as arguments. The result of combining a sequence of mappings $m_1, \ldots, m_n$ with one of the above operators *op* is a new mapping r whose domain is the union of the domains of m_1 through m_n. For any number i in r's domain, *r(i)* is the value obtained by applying *op* to the sequence of values $m_k(i)$ for all m_k that have i in their domains.

A.7.4 Field Access

If *id* is an identifier bound to an event or interval and f is one of its field names (a metric or an attribute) then *id.f* evaluates to the value of the field.

A.7.5 Time Units

Numeric values are unitless. Times computed using the *elapsed* function are also unitless as values, but they represent a time value in some time unit specific to the implementation. In order that a specification writer may use these time values in a sensible way (e.g., compare them to literals) operators are provided to convert literals in specified time units to their equivalent values in internal time units. If n is a numeric literal, then any of the following operators can follow n in an expression:

us = microseconds
ms = milliseconds
sec = seconds
min = minutes
cyc = cycles

For example, *10 ms* evaluates to the real number of internal time units equal to 10 milliseconds.

A.7.6 Arithmetic Operations

Some arithmetic operations are overloaded to work with triples and mappings as well as numbers. The operations on numbers and triples are described here. The section on mapping expressions describes operations on mappings.

Numeric operations. The numeric operations are: – (unary), – (infix), +, ∗, /, *div*, *mod*, *max*, *min*, *log*, *power*, *abs*, and *trunc*. The first five of these are defined as in Modula-3.

div and *mod* are infix operations whose arguments must be integral values. They produce integral results and are defined as in Modula-3.

min and *max* are invoked as functions, each taking two numeric arguments and returning a number. *min* returns the minimum of its arguments and *max* returns the maximum.

If a and b are numeric expressions, then the logarithm to the base a of b is written *log(a,b)*, and a raised to the b power is written *power(a,b)*. Both of these operations return numbers.

If n is a numeric expression, then *abs(n)* is the absolute value of n. *trunc(n)* returns the greatest integral number that is at most n for n positive, and the smallest integral number that is at least n for n negative.

Operations on triples. In what follows let t and u be triples of the form $[v, p, m]$. The notations $t.v$, $t.p$, and $t.m$ refer to the components of triple t. Note that a number n can be represented as the triple $[n, 0, 0]$. Arithmetic for mixed triples and numbers (except for the *log* and *power* operations) is defined first to convert the numbers into the corresponding triples, and then to use triple arithmetic.

The arithmetic operations on triples are defined in Figure A.2. The definitions are derived using the notion of a triple as a representation of a range of values with a "preferred" value. The v component of the result is the operation applied to the v components of the operand triples. The p component of the result is defined so that $v+p$ for the result is the maximum possible value that could result from applying the operation to values in the ranges of the operands. Similarly, the m component is defined so that $v-m$ for the result is the minimum possible value for the operation, treating the operands as ranges.

A.7.7 Relational Operations

The relational operations are: <, <=, >, >=, =, and != (not equal). These are defined both on numbers and on triples, and the result is a boolean. Their definitions for numbers are as expected. An expression of the form "*a op b op c*," where *op* is a relational operator, is equivalent to the expression "*a op b & b op c*." The definitions on triples, assuming t and u are triples, are as follows:

$$
\begin{aligned}
t + u &= [t.v + u.v, t.p + u.p, t.m + u.m] \\
-t &= [-t.v, t.m, t.p] \\
t - u &= t + (-u) \\
t * u &= [t.v * u.v, \max_{i \in t, j \in u}\{i * j\} - t.v * u.v, \\
&\quad t.v * u.v - (\min_{i \in t, j \in u}\{i * j\})] \\
1/t &= \text{if } 0 \in t \text{ then error} \\
&\quad \text{else } [1/t.v, \max\{1/(t.v + t.p), 1/(t.v - t.m)\} - 1/t.v, \\
&\qquad 1/t.v - \min\{1/(t.v + t.p), 1/(t.v - t.m)\}] \\
t/u &= t * 1/u \\
min(t, u) &= [min(t.v, u.v), \\
&\quad min(t.v + t.p, u.v + u.p) - min(t.v, u.v), \\
&\quad min(t.v, u.v) - min(t.v - t.m, u.v - u.m)] \\
max(t, u) &= [max(t.v, u.v), \\
&\quad max(t.v + t.p, u.v + u.p) - max(t.v, u.v), \\
&\quad max(t.v, u.v) - max(t.v - t.m, u.v - u.m)] \\
log(b, t) &= [log(b, t.v), log(b, t.v + t.p) - log(b, t.v), \\
&\quad log(b, t.v) - log(b, t.v - t.m)] \\
power(b, t) &= [power(b, t.v), power(b, t.v + t.p) - power(b, t.v), \\
&\quad power(b, t.v) - power(b, t.v - t.m)] \\
power(t, b) &= [power(t.v, b), power(t.v + t.p, b) - power(t.v, b), \\
&\quad power(t.v, b) - power(t.v - t.m, b)]
\end{aligned}
$$

Figure A.2: Arithmetic operations on triples. t and u are triples. b is a number. For *log* and *power*, $(t.v - t.m)$ must be greater than 0. The notation $i \in t$ means $t.v - t.m \leq i \leq t.v + t.p$.

$$t = u \equiv (t.v - t.m \leq u.v + u.p) \wedge (t.v + t.p \geq u.v - u.m)$$
$$t \; != \; u \equiv !(t = u)$$
$$t > u \equiv t.v - t.m > u.v + u.p$$
$$t < u \equiv t.v + t.p < u.v - u.m$$

A.7.8 Logical Operations

The logical operations are: & (and), | (or), ! (not), and => (implication). They have their usual meanings applied to boolean-valued arguments. & and | evaluate all of their arguments.

As described in the section on mappings, & and | are overloaded to work for mappings as well.

A.7.9 Operations on Intervals

If i is an interval whose start and end events are both timed, then *elapsed(i)* returns a triple representing the elapsed time for interval i. The p and m components of the triple are determined based on the units in which the timestamps are measured. If the units are cycles, the p and m components are zero. For any other units, they are one if the t component is non-zero. If the t component is zero, the p component is one and the m component is zero. If either the start or end event of i is not timed, it is an error to evaluate *elapsed(i)*.

A.7.10 Operations on Events

If e is a timed event, then *timestamp(e)* returns the value of e's timestamp, which is a number. If e is not timed, evaluating *timestamp(e)* is an error.

A.7.11 Aggregates

An aggregate expression describes the combination of a sequence of values to produce a result value. The sequence of values is produced by introducing a new identifier that gets bound to a series of values in a specified range and evaluating a specified expression (which may use the identifier) for each binding. The sequence resulting from evaluating the expression for each binding is then combined using a specified aggregate operator.

The range may be specified in two ways: it may be a sequence of events or intervals of a specified type and satisfying a specified predicate, or it may be the values in the domain of a mapping that satisfy a specified predicate.

We define an implicit component of a scope, called the *current log sequence*, that is used in evaluating aggregate expressions that range over events and intervals. In the global scope, the current log sequence is all events and intervals in the log. Within the scope of an interval declaration, it is all events and intervals wholly contained between (and not including) the start and end events of the interval being declared. For the purpose of defining the current log sequence for intervals, intervals are ordered by their end events. If two intervals have the same end event, they can appear in either order in an interval sequence.

The first form of aggregate expression (ranging over the current log sequence) is written:

{*op id : idtype* ***where*** *epred : expr*}

where *op* is an aggregate operator, *id* is an identifier, *idtype* is an event or interval type name, *pred* is a boolean-valued expression, and *expr* is an expression. *pred* and *expr* may use *id* but may not reference any identifiers bound by outer aggregate expressions. The *where* clause may be omitted. The values bound to *id* are those events or intervals in the current log sequence that have type *idtype* and for which *pred* is true.

The second form of aggregate expression (ranging over the domain of a mapping) is written:

{*op id* ***in*** *domain(m)* ***where*** *pred : expr*}

where *op*, *id*, *idtype*, *pred*, and *expr* are as above, and *m* is a mapping-valued expression. The *where* clause may be omitted. The values bound to *id* are those values in the domain of *m* for which *pred* is true. The domain values are produced in an arbitrary order.

The aggregate operators are:

$+ * \& \mid$ *min max mean stdev var the last first count*

count is special—the *: expr* is omitted from the aggregate expression, and the result is the number of different values to which *id* gets bound. (*count* is provided for convenience and readability. The same result is produced using the + operator and letting *expr* be the constant 1.) The other aggregate operators are defined to work on mapping-valued expressions as well as on non-mapping values.

For non-mapping values, the operators are defined as follows. The definitions of the first six operators ($+,*,\&,|,min,max$) are simply extensions of

op	empty seq.	single element v
+	0	v
*	1	v
&	*true*	v
|	*false*	v
the,min,max,last,first,mean	error	v
var,stdev	error	error
count	0	1

Figure A.3: Boundary case results for aggregate operators

their definitions for two arguments. If $a_1, \ldots, a_n$ is a sequence of values and *op* is one of the operators then $(a_1\ op\ (a_2\ op\ (\ldots\ op\ a_n)))$ is the result of combining the sequence values with the operator. (In other words, these six operators are reduction operators.) The result when the sequence is empty or has only one element is defined in Figure A.3.

The operators *mean*, *stdev*, and *var* compute the arithmetic mean, standard deviation, and variance, respectively, of the sequence values. For a sequence $a_1, \ldots, a_n$ of numbers these are defined as:

$$mean(a_1, \ldots, a_n) = \frac{1}{n} \sum_{i=1}^{n} a_i$$

$$var(a_1, \ldots, a_n) = \frac{1}{n-1} \sum_{i=1}^{n} (a_i - \overline{a})^2$$
where $\overline{a} = mean(a_1, \ldots, a_n)$

$$stdev(a_1, \ldots, a_n) = \sqrt{(var(a_1, \ldots, a_n)}$$

(The variance formula is what statisticians would call a *modified sample variance* or *unbiased estimate* of the variance). The result of applying one of these operators to a sequence of triples is a triple whose components are computed using the above formulas.

The operators *first* and *last* evaluate to the first and last values in the sequence, respectively.

The operator *the* applied to a single-element sequence returns the single value. It is an error to apply *the* to a multi-element sequence.

Finally, when the sequence values are mappings, the aggregate operator defines how to combine the mappings. The result of combining a sequence of

mapping values with an aggregate operator is a new mapping whose domain is the union of the domains of the mapping values. The value to which the new mapping maps a domain element i is the result of combining the values to which i is mapped by the mappings in the sequence using the aggregate operator as described above.

A.8 Grammar

In the grammar that follows, italicized words are non-terminals, *term*,+ means one or more occurrences of *term*, separated by commas, and *term*,* means zero or more occurrences of *term*, separated by commas. (*term*;+ and *term*;* are defined similarly). Comments are preceded by **%** and extend to the end of the line.

spec ::= `perfspec` id *stmt*;+ `end` id
stmt ::= `def` *constdef*;+
| `solve` *solvedecl*;+
| [`timed`] `event` *eventdef*;+
| [`nested`] `interval` *intervaldef*;+
| `assert` *expr*;+
constdef ::= id `=` *expr* | id `=` `?`
solvedecl ::= [`data` *varrange* `:`] *expr* [`,` `var` id [`,` `cor` id]]
eventdef ::= id `(` id,* `)`
intervaldef ::= id `=` *intervalhead* [`metrics` *metricdef*,+] `end` id
intervalhead ::= *varrange* `,` *varrange*
metricdef ::= id `=` *expr*
expr ::= *expr op expr*
| `-` *expr*
| `[` *expr* `,` *expr* `,` *expr* `]`
| *expr* `->` *expr*
| *expr* `(` *expr*,* `)`
| *aggrexpr*
| `(` *expr* `)`
| *const*
| id
| *expr* `.` id
aggrexpr ::= `{` *aggrop varrange* [`:` [*expr*]] `}`
varrange ::= id *varset* [`where` *expr*]
varset ::= `:` id | `in` *expr*

aggrop ::= `+` | `*` | `&` | `|` | `count` | `mean` | `stdev`
| `var` | `max` | `min` | `the` | `last` | `first`
op ::= *arithop* | *relop* | *boolop*
arithop ::= `+` | `-` | `*` | `/` | `div` | `mod`
relop ::= `<` | `<=` | `>` | `>=` | `=` | `!=`
boolop ::= `&` | `|` | `!` | `=>`
const ::= num | num *timeunit* | `true` | `false`
timeunit ::= `us` | `ms` | `sec` | `min` | `cyc`