Rule-Based Runtime Verification in a Multicore System Setting

A dissertation submitted to the University of Manchester for the degree of MSc in the Faculty of Engineering and Physical Sciences

2010

Giles Reger

School Of Computer Science

Contents

Word Count : 28,824

List of Figures

List of Tables

Abstract

Runtime monitoring is a useful approach to program verification; less complicated and resource dependent than model checking yet more powerful and complete than program testing, runtime monitoring can be used to increase the confidence in the correctness of programs as well as altering the execution of a program in response to bad behaviour.

We have begun to reach the limits of what can be achieved by attempting to increase the efficiency of single processor chips. Therefore to increase the performance of programs in the future the concurrency offered by multicore machines must be harnessed.

This study looks at whether rule-based runtime verification can benefit from being used within a multicore system setting, and if so how. To do this an experimental framework and an optimised monitor are constructed, and evaluated using a number of microbenchmarks and selected benchmarks from the DaCapo benchmark suite.

Results from the microbenchmarks are reasonably positive but indicate that monitored properties must be very complicated or workloads very large to see the benefits of parallel execution. This is seen in the monitoring of the DaCapo benchmarks where only simple data-structure related properties were monitored - as a result little improvement was seen.

The main conclusion of this study is that unless usefully complex properties are developed the best approach to harnessing the concurrency offered by a multicore system is to act outside of the monitor itself to decompose the execution trace, so that a number of monitors can evaluate these smaller traces in parallel. Typically, during *online* monitoring only a small amount of work is completed on each step, therefore a parallelisation of the internals of the monitor is rarely effective.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning

Notice of Copyright

Acknowledgments

Firstly, I would like to take this opportunity to thank my supervisor Professor Howard Barringer for introducing me to the exciting field of runtime monitoring and giving a lot of greatly appreciated advice and guidance through this project. And secondly, I would like to thank Clare, for all of her encouragement, endurance and support over the past few months.

Chapter 1

Introduction

This project examines whether rule-based runtime verification can benefit from being used within a multicore system setting, and if so how. This chapter presents the motivations behind this project, the overall aims and objectives of the project and the project's achievements, finishing with an overview of the rest of this document.

1.1 Motivations

Ultimately the main task carried out by anybody working in of the computer industry and associated communities, is the solving of certain problems using computers. This will inevitably involve the development of computer systems, both in hardware, software and a combination of the two. For a system to be a valid solution to a problem it must firstly actually solve the problem correctly and secondly it must solve the problem whilst the problem is still relevant, that is it must complete within a reasonable time.

Therefore there must be mechanisms in place to establish confidence in a system's correctness but to be used these mechanisms must be both efficient and usable. There exist many methods for establishing some level of confidence in a program's correctness however more complete methods, such as *model checking*, are often too costly and less complete methods, such as *program testing*, are used. As explained in chapter 2 the field of *runtime monitoring* exists between the two and is the focus of this work.

Runtime monitoring represents a compromise between program testing and heavyweight formal methods such as model checking. Generally program testing requires a program to be executed multiple times with different inputs and the more times the program is run more confidence can be had in its correctness - the time taken to complete this process will therefore be the time to execute the program multiplied by the number of times it is run. Model checking involves the construction of a model representing all possible runs of the program followed by a complete inspection of this model checking that the property being checked holds on all runs - this will typically take much longer than the simple running of the program and requires a lot of time to set up.

Ideally runtime monitoring will be closer to program testing than model checking in runtime however it must carry out extra computation on top of the simple running of the program. To bring runtime monitoring closer to program testing in running time this extra computation must be addressed.

In the past if programmers wanted increased performance all they had to do was wait for the next generation of faster chips to be released. However, as Herb Sutter writes in [59], the 'Free Lunch' of ever increasing processor speeds is over. It is widely agreed that the future is in *multicore computing* - placing many cores on a single processor chip. Previously parallel programming efforts were largely restricted to high performance of scientific computing communities but with multicore chips finding their ways to desktop and laptop computers it is becoming more and more important to harness this concurrency.

Therefore this project aims to explore how the concurrency offered by multicore systems can be applied to the rule-based runtime monitoring tool RuleR.

1.2 Aims and Objectives

This project looks at how RuleR can be developed to take advantage of multicore machines. To this end the project has the following aims

A Explore architectural and algorithmic ways to

i Increase the *performance* of, and

ii Decrease the *interference* of

the RuleR runtime monitoring tool through the use of multicore machines.

B Demonstrate scope for improvement through practical experimentation.

Here the term *interference* has two meanings - firstly any side-effects of the monitoring process which may alter the behaviour of the monitored program should be reduced, and secondly the time to run the monitored program and time to complete the monitoring process should be considered separately and the slowdown of the monitored application reduced where possible. This concept is only relevant when considering online monitoring (carrying out the monitoring process whilst the program is running), rather than offline monitoring (saving the observed execution trace to file and running the monitor afterwards) - but as only the use of RuleR as an online monitoring tool is considered in this project this concept remains relevant throughout.

The first aim focuses on two goals. Firstly it is concerned with finding ways to carry out runtime verification more efficiently. This is important as applications are increasingly becoming larger and more complex. Also for a tool to be used and accepted it must maintain reasonable efficiency. Secondly it is concerned with reducing the interference of the monitor with the monitored application. This is important to allow certain real-time properties to be monitored, it will also increase efficiency.

My second aim focuses on the need to demonstrate that any idea, to be worth pursuing, has the potential to make a difference. In a project of this size it is not possible to carry out extensive studies or field tests or reviews, this demonstration therefore must be done through experimentation and careful analysis of results.

To give substance to these aims the following objectives set out how these aims are to be achieved;

1. Create a number of conceptual architectures and refine them through prototyping,
2. Implement these architectures within a coherent framework,
3. Evaluate these architectures for *efficiency* and *interference* through both microbenchmarks and performing runtime monitoring on a real-world program,

These objectives were followed through and relate to chapters 4, 5 and 6 and 7 respectively.

1.3 Contributions From This Study

This study has produced an experimental framework and an optimised RuleR monitor, both of which can be used for the runtime monitoring of programs. The framework implements a number of different approaches to communicating with the monitor with the aim of increasing the performance and decreasing the interference of the monitoring process. As the name suggests the framework is a structure in which a number of different approaches can be explored and there remains further scope for refining the approaches developed within it. The optimised monitor has been extended to offer a parallel mode where part of its work is carried out concurrently, as well as some work factored out into a cleaner thread running in parallel. The monitor can successfully be used without the framework.

A large amount of time of this study was used to set the ground for the experimentation seen here - both in the creation of the experimental framework and optimised RuleR monitor and in the development of microbenchmarks and organisation of the DaCapo benchmark suite. As mentioned in Chapter 8 there are many further experiments which can be carried out and there is more to be learned from the setup developed in this study.

1.4 Roadmap

The rest of this dissertation will be organised in the following way

Runtime Monitoring - This chapter introduces the field of runtime monitoring. A selection of runtime monitoring tools are presented and RuleR is examined in detail.

Utilising Multicore - This chapter discusses the field of multicore machines, as well as discussing relevant areas of parallel software development and outlining the experimental machine used in this project.

Analysis - This chapter gives an analysis of the problem to be solved and possible solutions. The current overheads involved in the monitoring process are discussed and ways to reduce these are suggested.

An Experimental Framework - This chapter describes the experimental framework and optimised parallel RuleR monitor developed for this project.

Experimental Results - This chapter presents some results from a number of microbenchmarks designed and ran to explore the limitations and scope of the developed framework and monitor.

Evaluation - This chapter uses the international DaCapo benchmark suite to evaluate the tool, examining a number of data-structure orientated properties.

Conclusion - This chapter finishes the dissertation by discussing to what extent the aims and objectives of the project were met and suggesting possible future work.

Chapter 2

Runtime Monitoring

This chapter gives the relevant background in runtime monitoring, including an in-depth description of the rule-based runtime verification tool RuleR. The history of the field is presented leading into a discussion of other runtime monitoring tools.

2.1 Runtime Verification

This section introduces the field of runtime monitoring and verification from a historical perspective. First the roots of the field are explored before a discussion of its current state is given.

2.1.1 A Short History of Program Verification

Since computer programs began to be written in the late 1940s there has been a need to ensure that they operate as required. To do this a specification of correct behaviour is required. Program testing runs the program and detects deviation from the specification. Formal verification formalises the specification and applies formal methods to verify this against a model of the program.

Dijkstra famously said that "Testing shows the presence, not the absence of bugs", program testing increases confidence in the program behaving as specified, but can not prove or demonstrate correctness. Formal verification gives more concrete guarantees about correctness provided an appropriate model and specification are provided, and this is often very difficult to get right and it may be the case that a proof cannot be constructed.

Formal Verification

The field of formal verification of programs can be traced back to 1949 when Alan Turing published a paper 'Checking a large routine' [60]. This was followed by 'Assigning Meanings to Programs'[28] published in 1967 by Robert Floyd, which looked at verifying flow charts. In 1967 Edsger Dijkstra [23] introducing the concept of program refinement (the generation of programs from these specifications). And in 1969 Tony Hoare published work on developing an axiomatic approach to the program specification and verification problem [34].

In 1962 John McCarthy wrote a paper titled 'Checking mathematical proofs by computer' [46] in which he discussed how computers could help mathematicians construct complex proofs. In 1967 McCarthy also published the first paper proving the correctness of a compiler [45] for arithmetic expressions, this laid the foundations for the concept of verifying compilers.

Then in the 1970s and 1980s important work was carried out in the area of specification languages. In 1977 Pnueli published his seminal paper on temporal logics in relation to computer science[1][54] leading to Linear Temporal Logic (LTL), one of the more popular specification logics for concurrent system properties. In 1972 the Vienna Development Method was introduced by IBM, [62]. In 1977 a language called Euclid [41] was designed at the Xerox PARC lab for writing verifiable programs. In 1980 Z notation was presented by Abrial [8]. Finally in 1982 the temporal logic Computation Tree Logic (CTL)[2] was introduced as an alternative to LTL, [26]. These specification languages paved the way for automated formal verification techniques.

By this time the area of computer-aided proof had come on a great deal. Previously in 1972 Robin Milner working at Stanford University (with McCarthy) had developed the LCF (Logic of Computable Functions) proof-checking system. This was improved and extended giving the Edinburgh LCF in 1979 with Michael Gordon [31] and Cambridge LCF [50] in 1987 by Larry Paulson. Gordon went on to develop the theorem prover HOL, based on Cambridge LCF and focused on hardware verification, which led Paulson to develop the interactive theorem prover Isabelle [51]. This work led to the development of PVS (Prototype Verification System) [49] by the Stanford Research Institute in 1992, and Coq in 1994, among others. These tools are interactive, requiring the user to form theories in complex logics and help the prover 'steer' the proof.

Theorem provers for higher ordered logics need to be interactive to an extent due to their complexity. However theorem provers for FOL (First-Order-Logic) and other logics of similar and reduced complexity can be fully (or mostly) automated. The development of Resolution by John Alan Robinson in 1965 laid the ground for more fully automated systems, such as the modern automated theorem provers Vampire [56] and SPASS [7].

The development of temporal logics and their relation to state machines (Kripke structures) saw the evolution of work in the field of *Model Checking*, which emerged as the major player in the field of formal verification. A key area in which model checking enjoyed great success was that of hardware verification.

SPIN [5] is an example of a widely used model checker designed for verifying asynchronous distributed algorithms , first released in the 1980s. A more recent example is the JavaPathFinder model checking tool for Java programs [61], first released in 2003, which uses a VM, consisting of a state generator and state explorer, running on top of a JVM, to verify Java bytecode.

In 1987 Bertrand Meyer formalised the concept of *Design by contract*(DbC) in the Eiffel programming language, [47], [48]. DbC is the application of pre and post conditions, and program invariants, to the object orientated paradigm. The contracts are specified formally and then checked against the program.

[1]Temporal Logic had been previously introduced to mathematics by Arthur Prior in his work Time and Modality published in 1957

[2]CTL is a branching logic, rather than a linear logic, and is a subset of the modal μ logic. Both CTL and LTL are a subset of CTL* and share a common subset but are not equivalent.

Other more recent implementations of this concept include Spec#, Jahob and the Java Modeling Language (JML).

In 1998 Amir Pnueli introduced the notion of *translation validation* [55] as an approach to the verification of compilers. The idea being that each compilation is followed by a validation stage which verifies that the produced target code correctly implements the source code. The concept of a completely sound verifying compiler is an important goal to the field of computer science, as reiterated by Hoare in 2003 when he presented it as an achievable Grand Challenge [36].

Program Testing

Alongside these efforts in formal verification of programs people were writing and debugging programs by other means. In 1966 Evans and Darley published a survey looking at the then current methods for on-line debugging techniques[3] [27]. These techniques consisted mainly of stepping through the execution of the program for given inputs and inspecting the intermediate values of program variables or registers. This is still one of the basic approaches of program testing.

Even with many advancements in the area of formal program verification the main method for increasing confidence in a program's behaviour remains program testing. The industry of software engineering has been the main source of testing methodologies. Program testing has become a wide field, which has been divided in different ways including

- Testing from different perspectives - white box and black box testing.
- Testing at different levels - e.g. unit testing, integration testing, system testing.
- Testing different attributes - e.g. functional testing, load testing, usability testing.
- Testing with different aims - debugging, demonstration, destruction, evaluation, prevention [4].

More recently, in the 2000s, a move in the area of Agile Development has introduced the concept of Test Driven Development [18], where the tests are written before the code and 'drive' the development by forming a description of the desired functionality.

As with program verification, the process of program testing was soon automated, and integrated into development environments allowing the creation of *test harnesses*. Now tools exist allowing programmers to run large test suites at the click of a button and carry out *continuous integration* of new code.

It is important to note that program testing and formal program verification are not at odds, in fact they complement each other, as noted by Tony Hoare in [35] - program testing provides a framework for development allowing programmers to shape, explore and communicate ideas, however formal program verification allows for a great deal more confidence in the correctness of a program.

[3] on-line being whilst the program is running/through executing the program, rather than off-line by static inspection. This was in response to a move from batch processing to interactive processing made possible by time sharing systems

[4] These testing aims come from a 1988 report by Gelperin and Hetzel [30].

2.1.2 A Definition of Runtime Verification

There are a number of forms of runtime monitoring. They all monitor a single execution of a program, in the same way as testing, therefore carrying out runtime monitoring on many runs of a program will increase the confidence in the correctness of that program.

The different forms of runtime monitoring act on the execution in the following ways. *Runtime Verification*(RV) is the process of monitoring the execution to decide if it conforms to a specification. *Reactive Monitoring* is the process of altering the execution in response to violation of a specification, and is a form of "program steering". *Enforcement Monitoring* is the process of only allowing the execution to progress in accordance with a specification. The second two obviously depend on the first.

Runtime Verification is related to *Model Checking*. Model checking is the process of automatically checking whether a program meets a given specification, this means for all possible runs of the program the model checker computes whether the program behaves as specified. Formally, given a program represented by a structure M with an initial state s and a specification ρ, the model checking problem can be expressed as $M, s \models \rho$ - that is in the model M the property ρ holds for state s.

But the model checking problem is intractable for languages as expressive as programming languages. The structure M may describe an infinite number of possible runs of the program, each with a possibly infinite number of states, suffering so called *state explosion*. Abstraction must be applied to reduce this to a manageable, or even finite, size. Model checkers are often used as debuggers, rather than verifiers, exploring the structure M to a certain depth searching for errors.

To reduce complexity model checking applies abstraction methods to reduce the number of considered runs. Runtime verification takes this abstraction to the extreme by only considering a single execution trace of the program, a single path through the structure M. This represents a move from formal program verification toward program testing. As only a single execution trace is examined the result is much weaker than formal program verification, but stronger than program testing. Runtime Verification might be described as a *lightweight formal method*.

Runtime verification has a number of advantages over formal program verification;

- There is more information available at runtime than from static analysis of the code,
- Rather than checking a model or abstraction of the code the actual code is being verified,
- Action can be taken immediately if an execution is seen to violate a specification.

The process consists of three main elements

- Formal specification of the property to be monitored in a *specification language*
- Instrumentation of the application to be monitored to access the relevant information.
- Monitoring the output from the instrumentation, this involves choosing how to deal with the data-free (propositional) case and how to handle data values

2.2 Runtime Verification Tools

The field of runtime verification is not a new one, but may not have always gone by this name and the focus has shifted over the years. As mentioned previously the very first debugging systems examined the intermediate properties of a program's execution at runtime. Modern runtime verification tools automate this process and use formal specifications to describe the behaviour being checked.

In the early 1980s Bernhard Plattner carried out a survey looking at the monitoring of program execution [53] he concluded that at the time it had two main purposes - performance evaluation and debugging. In the last 30 years both of these areas have evolved into complex and active fields of their own. More recently the purposes of runtime monitoring have been extended to include areas such as intrusion detection and program steering.

One aim of this project (A.ii) is to reduce the interference of the RuleR tool, again this is not a new goal. Plattner called this approach real-time monitoring, focusing on avoiding breaking real-time constraints of the application, his PhD introduces a technique for real-time monitoring [52]. He implements a real-time monitoring system for a simple high level, block structured programming language. An interesting conclusion he draws is that for real-time monitoring the response time should if possible be constant and at least independent of the input size.

In 2001 a series of workshops focusing on the area of runtime verification [4] was initiated, interested in bringing together work on how to monitor, analyse and guide the execution of programs. Of course due to this being an important area in the field of computer science many relevant pieces of work also exist in other conferences and journals. Since 2001 we have seen an acceleration in work in this area.

One contributing factor to this acceleration may be an advancement in the field of program instrumentation. The process of program instrumentation can be a difficult one, often requiring the manual insertion of assertions into the code or machine code directly. A recent development that greatly helped is that of AOP (Aspect Oriented Programming) first, and still most notably, found in the tool AspectJ (see 2.3.4) developed from 2000 by Xerox PARC and later as Eclipse AJDT. AOP allows easy instrumentation of programs by *weaving* specified code, representing crosscutting concerns, into the program at specified points.

2.2.1 Existing Runtime Verification Tools

There have been a number of specific runtime verification tools. Here I discuss a selection of tools in rough chronological order.

Java-MaC [44] This tool is based on the Monitoring and Checking (MaC) framework. The MaC framework uses two specification languages - MEDL and PEDL. MEDL (Meta-Event Definition Language) is similar to PT-LTL (Past-Time Linear Temporal Logic) with timing operators and is used to specify properties to monitor. PEDL (Primitive Event Definition Language) is used for instrumentation. PEDL scripts define the MEDL events and conditions in terms of system objects, and this mapping is used to generate an event recogniser and observation filter.

TemporalRover[24](TR) This is the only well-known commercial runtime verification tool, first appearing in 2000 developed by Doron Drusinsky. It can be used to verify applications written in C, C++, Java, Verilog and VHDL, using specifications written in LTL or MTL (Metric Temporal Logic) augmented with additional operators. TR assertions are written as comments in the application, which are expanded into source code by the TR parser. The ATG Rover, a related tool, can be used to generate test sequences from the specifications, a form of Model Based Testing. More recently Drusinsky has focused on using UML as a specification language.

Jass (Java with assertions) [1][16] This is a precompiler supporting extended assertions for Java. Jass provides pre and post conditions and loop and class invariants, among other features. Importantly it also provides trace-assertions, based on the process algebra CSP (Communicating Sequential Processes), where the trace is defined by beginnings and ends of method invocations.

JavaPathExplorer [33] (JPaX) This tool has been developed by NASA to verify code related to their Mars mission. The tool performs both logic based monitoring and error pattern analysis for common types of errors. Logics are expressed in Maude [3](a term rewriting logic). An instrumentation script specifies how the Java bytecode is to be instrumented by Jtrek - a Java bytecode engineering tool. JavaPathExplorer is related to the JPF (JavaPathFinder) tool.

tracematches [9] This tool extends AspectJ with regular expressions over the computation trace (AspectJ pointcuts can only refer to current state). Joinpoints are specified in a similar way to in AspectJ but now a tracematch can be declared as a regular expression over joinpoints. This tool is very flexible, combining the code instrumentation, specification language and monitoring algorithm and extending the source language with them. In tracematches the trace is defined as entries and exits from joinpoints. Interestingly if a tracematch matches multiple times (through different variable bindings) the advice will be executed for each matching. In the presence of a multithreaded program the user can decide to either consider the trace of each single thread separately, or the interleaved trace of all threads.

Java-MOP [21] This is a tool based on the Monitoring-Orientated Programming technique. Java-MOP supports both DbC-like specification languages, such as JML and JASS, and trace languages, such as regular expressions and LTL (and variants of). For extra flexibility specification logics can be added by use of logic plugins. Instrumentation of the application is through AspectJ - synthesized monitors are wrapped as advices and then weaved into the application.

RuleR, the tool at the focus of this project, has its roots in the METATEM and EAGLE tools.

METATEM [11] is an executable temporal logic. LTL formulas are separated into a boolean combination of pure past, present and pure future parts. Depending on the the past part the state of the current moment is built from the present part and obligations are generated from the future part. In this way the METATEM interpreter builds traces state by state.

EAGLE [12] [15] is a rule-based framework for defining and implementing finite trace monitoring logics. The EAGLE logic is a restricted first order, fixed-point, linear-time temporal logic with chop over finite traces. The monitoring algorithm operates on a state-by-state basis, as opposed to storing the execution trace, this avoids the need for backtracking.

The EAGLE monitoring framework has been implemented in Java. The framework compiles the specification file to generate a set of Java classes representing the rules, which are then compiled into bytecode. This produces a list of monitors, which are evaluated for each observation from the trace to generate a new list of monitors.

EAGLE is a very expressive and powerful logic, however it is not necessarily that efficient. The **RuleR [12], [14]** tool has been been developed as a practically useful and efficiently executable subset of EAGLE. RuleR is a low-level conditional rule based system and is described further in section 2.3.

More recently there have been many new tools entering the scene, these tools are extending the application of runtime verification tools by working on domain-specific problems or new languages. Here is a quick overview of a few recent advancements.

- Introduced in 2007 the **ARVE [57]** tool from the Toshiba Corporation focuses on the C/C++ platform, whereas most tools here have focused on Java. ARVE uses a symbolic debugger to carry out aspect-orientated runtime verification based on a property described by a deterministic finite automata.

- Introduced in 2008 the **Tamago [19]** platform from Belhaouari and Paschanski enforces formal contracts, written as logical assertions or state transition systems, for component based systems. It provides various different percolation patterns for introducing contracts. The platform does not use code injection, but uses a container-based architecture.

- Around this time the **ORCHIDS [32]** tool appeared.The ORCHIDS tool is designed specifically for intrusion detection. The tool uses predefined patterns to detect anomalous and malicious behaviour in the program trace, and then takes appropriate action.

- Introduced in 2009 the **DMaC [64]** tool, standing for Distributed Monitoring and Checking, builds on the MaC framework to provide a tool for specifying and verifying distributed network protocols. The DMaC tool uses a variant of the MEDL specification language from the MaC framework.

- Also introduced in 2009 was the **LIME [39]** tool from Helsinki University of Technology, aimed at extending the DbC approach to behavioral aspects of interfaces. It uses past and future time LTL, NFA or regular expression properties to monitor the interaction of software components through interfaces for Java programs.

As well as the pure runtime verification approach some static verification systems also have runtime-checkers. Such as the previously mentioned JML [43], Jahob [63] and Spec#.

2.2.2 Specification Languages

All of the above tools use a language separate from the implementation language as a specification language, this allows a separation of the RV framework from the implementation. This may also make specifications harder to write, an extra language to learn for the user and parsing and manipulating work for the implementor. However the framework is not tied down to one particular implementation language[5].

It is interesting to consider the range of logics and languages the above tools use to specify desired (or undesired) program behavior. For state assertions forms of propositional or, in some cases, first order logic are adequate, but for trace assertions other languages are required. Many of the above tools extend or adapt LTL. Some of the other specification languages used include finite state machines, rule systems and regular expressions.

One problem found with the use of standard LTL for runtime verification is that it is defined over infinite traces, and in practice at runtime a tool will deal only with finite traces. The approach RuleR takes is that of a four-valued logic[6], allowing properties to be at any one point either *true, false, still_true* or *still_false*.

Many of the above tools, including RuleR, allow specifications written in one language to be translated into the native specification language of the tool. Some tools include plugins allowing the user to load, or define, domain specific logics, for example Java-MOP and JavaPathFinder.

2.2.3 Taxonomy of Tools

In 2004 Delgado et al. published 'A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools' [22] aiming to classify runtime monitoring systems. The dimensions they give for defining a runtime monitoring tool are

- *Specification Language* - Is the specification language an extension of the source language or separate? What level of abstraction does the language offer? What types of properties can be expressed? At what level can properties be specified?
- *Monitor* - Is the monitor online or offline, could it be either? Is instrumentation manual or automated? Does the monitor share the resources of the application?
- *Event Handler* - Can the monitor affect the application's behavior? What actions can be taken in response to a violation, can different kinds of violation be differentiated?
- *Operational Issues* - What source languages can the tool be applied to? Is the tool dependent on certain hardware or software? How mature is the tool?

[5] Although as many tools only possess implementations in single languages tying a tool down to one implementation language would not be too different from the current situation.

[6] In fact the RuleR logic is more accurately five_valued as it also allows a property to be unknown. A similar four_valued logic is described in [17] as RV-LTL

Their taxonomy is geared towards the use of the tools, rather than the design, and even in this direction I feel fails to analyse properly the range of properties monitorable by each tool. However some distinctions are important, for example placement of instrumentation code, and whether monitoring can be online or offline. What Delgado et al. mention but do not include in their taxonomy or synopsis of tools is the subject of usability, such as visualisation, ease of forming specifications, or automated tools for translating specifications.

The tools given in 2.2.1 have been roughly classified in table 2.1, this does not follow all the dimensions of the taxonomy presented in [22]. The classification presented here explores specification language, instrumentation, monitoring (algorithm), implementation features and usability. The details from the table have been extracted from literature on the tools, there was not enough time within this project to practically examine each tool.

2.2.4 Summary

The above tools represent extensive work in the area of runtime monitoring, so one might question whether yet another runtime verification tool is required. However I believe the work of this project is relevant for two reasons. Firstly the RuleR family of tools explores *rule-based* runtime verification systems, a unique avenue with great potential. Secondly this project is concerned with increasing the *usability* of the existing RuleR tool by exploring ways to increase the performance of the tool and therefore extend the number of programs the tool could be applied to.

One thing to learn from this examination is that no single unifying solution has yet presented itself. However I believe that two beneficial approaches have become clear, automated instrumentation and intuitive, usable, specification languages.

		Java-MAC	TemporalRover	JASS	JPaX	tracematches	Java-MOP	EAGLE	(Java) RuleR	LOGSCOPE
specification language	LTL variant	✓	✓		✓		✓			
	regular expressions					✓	✓			
	automata						✓	✓	✓	✓
	conditional rules							✓	✓	
	DbC (algebra)			✓			✓			
	Higher Level			CSP						
	plugins/translation				✓		✓			
x-value logic			3/4		2	2	2		5	5
instrumentation	manual	✓	✓	✓			✓			
	automated Separate Script	✓			✓		✓	✓	✓	
	automated Extension of Spec. Lang.					✓				
	Uses log files									✓
monitoring	online	✓	✓	✓		✓	✓	✓	✓	
	offline									✓
	verification	✓	✓	✓	✓	✓	✓	✓	✓	✓
	reactive	✓	✓	✓			✓	✓	✓	
Also uses pattern analysis					✓					
current implementation	Monitoring Java		✓	✓	✓	✓		✓	✓	
	Monitoring other languages		✓							✓
	Same thread as application		✓	✓						
	Different thread	✓	✓		✓		✓			
	Research	✓		✓	✓	✓	✓	✓	✓	✓
	Non-Research		✓			✓				✓
usability	GUI						✓			
	MBT tool?		✓							

Table 2.1: Categorising the tools presented in 2.2.1.

2.3 RuleR

RuleR is a rule-based runtime verification tool. It consists of a specification language and an algorithm. There also exists a Java implementation of the tool, which is the focus of this project.

2.3.1 Language

The RuleR language allows the user to define properties in terms of parameterised conditional rules. The language is very powerful and complex.

The basic building block of the RuleR language is the rule. A rule definition takes the form

$$rulename : antecedent \rightarrow consequent;$$

A rule states that if the antecedent is true for the current step then there is an obligation for the consequent to hold on the next step. These rules can also be parameterised

$$rulename(x_1 : \tau_1, x_2 : \tau_2 ...) : antecedent \rightarrow consequent;$$

Where the antecedent and consequent can make use of the variables x_1 etc. The binding of these variables is typical of a typed functional language.

There are then syntactical extras to the language to make writing rule systems easier. Rules can have a number of branches, on a single step more than one of these branches may fire. It is useful to think of each branch as a separate rule and the syntax a form of renaming

$$rulename\{antecedent_1 \rightarrow consequent_1; antecedent_2 \rightarrow consequent_2; ...\}$$

Antecedents can be nested, and this can be done in two different ways.

(1) $rulename\{antecedent_1\{|antecedent_{1a} \rightarrow consequent_{1a}; antecedent_{1b} \rightarrow consequent_{1b}; ...|\}...\}$

(2) $rulename\{antecedent_1\{: antecedent_{1a} \rightarrow consequent_{1a}; antecedent_{1b} \rightarrow consequent_{1b}; ... :\}...\}$

In the first way (1) the branches are executed in parallel, as the branches of a rule are. In the second way (2) the branches are executed in sequence and only the first branch able to fire does so, one can give a *default* branch by giving an antecedent of true[7] or using the work `default` as the last branch.

Rules can be preceded with modifiers, the modifier can be either *always, state* or *step*. Always means that the rule will always persist unless explicitly removed. State means that the rule will persist until it is fired. Step means that if the rule is not fired on the next step it will disappear.

$$modifier \;\; rulename : rulebody;$$

Rulenames and observations must be distinct. We can then consider atoms to be rulenames or obser-

[7] Or leaving the antecedent blank, recall the antecedent is a conjunction and an empty conjunction is true.

vations and literals to be positively or negatively occurring atoms.

Antecedents are conjunctions of literals, which can be parameterised, the binding works in a similar way as before. Given

$$R(a : int) : obs(x : int) \rightarrow R(x + a);$$

the x and a in the consequent binds to the x and a in the observation and rulename respectively.

Consequents are disjunctions of conjunctions of literals. The conjunctions of rulenames represent obligations for the application, disjunctioning these conjunctions represents *possible* alternative obligations that the application might fulfill to continue to validate the specification.

The specification defines a rule system with a step relation defined over configurations for the rule system. The language accepted by a rule system is the, possibly infinite, set of finite observation traces accepted by that rule system. An observation trace is said to violate a rule system if it is not in its language. The paper 'Rule Systems for Runtime Monitoring: from EAGLE to RULER' [15] presents formal trace semantics for RuleR.

2.3.2 Algorithm

The RuleR algorithm takes a RuleR specification, as described in 2.3.1, and an observation trace and decides a truth value from {`FALSE, TRUE, STILL_FALSE, STILL_TRUE, UNKNOWN`}. The algorithm is described in prose in Figure 2.1 and given more formally in Figure 2.2.

As previously stated a RuleR specification defines a rule system, the algorithm represents a breadth-first search of this rule system's possible configurations given the input observation trace. The frontier represents the configurations the monitor is in, if the frontier is empty the monitor has failed. From here on I use *state* to mean configuration.

The frontier is initialised by the initials set given by the user and the next frontier is computed iteratively. Each state in the frontier is considered in turn - the new observation is added to the state; the state is searched for rule activations with antecedents evaluating to true in that state; these rule activations are fired to create a set of new states. These sets of new states are unioned together to generate the new frontier. If a state is empty or does not contain a rule assertion from the assert set it is deleted from the frontier.

```
1  create an initial set of initial rule activation states
2  WHILE observations exist DO
3  -Obtain next observation state
4  -Merge observation state across the set of rule activation states
5  -raise monitoring exception if there's no self-consistent merged sate for each
   of the current and self-consistent merged states
6  -use activated rules to generate a successor set of activation states
7  -union successor sets to form the new frontier of rule activation states
8  OD
```

Figure 2.1: The RuleR Algorithm given in prose, taken from [15].

input : Initial, Accept, Forbidden, Assert sets of Initial, Accepting, Forbidden and Asserted Rule Activations respectively, and Trace a list of Observation Events

output: Has the trace met the specification in {`false, true, still_false, still_true, unknown`}

```
1  begin
2      frontier := Ini;
3      while Trace not empty do
4          Let obs = Head (Trace);
5          foreach state s ∈ frontier do
6              s := s ∪ obs;
7          newFrontier = ∅;
8          foreach state s ∈ frontier do
9              foreach ruleActivation ra ∈ s do
10                 get rule r = (antecedent → consequent) for ra;
11                 match consequent with ⋁ c_i;
12                 Bindings = unify(ra,r);
13                 foreach b ∈ Bindings do
14                     if b(antecedent) then foreach c_i ∈ consequent do
15                         newState = (s ∪ c_i)/ r;
16                         if r ∩ Assert = ∅ then newFrontier ∪ = newState;
17         frontier = newFrontier;
18         if frontier= ∅ then return False;
19         if frontier ∩ Success ≠ ∅ then return True;
20     if ∃s ∈frontier. s ∩ Forbidden == ∅ then return Still_True; else
21         if ∀s ∈frontier. s ∩ Forbidden ≠ ∅ then return Still_False; else
22             return Unknown;
```

Figure 2.2: RuleR algorithm

The truth value of the monitor at each step is calculated as follows

$$monitor = \begin{cases} true & \text{if a state in the frontier contains a rule activation from the success set} \\ false & \text{if the frontier is empty} \\ still\_true & \text{if there exists a state with no rule activations in the forbidden set.} \\ still\_false & \text{if there does not exist a state with no rule activations in the forbidden set.} \\ unknown & \text{otherwise} \end{cases}$$

As explained earlier this five-valued logic is required due to the evaluation of specifications over finite traces. Let us consider two common forms of properties - liveness and safety properties. A liveness property states that a property will eventually hold and a safety property states that a property holds. At the end of a finite trace if a liveness property has not yet been satisfied it can be thought of as still_false, and if a safety property has not yet been falsified it can be thought of as still_true.

By exploring the possible states in a breadth first manner all possible states of the monitor are explored concurrently. This means that when a state becomes false backtracking is not required.

2.3.3 Two Examples

This section gives two short examples to demonstrate the use of RuleR. Further examples are given in appendix A.

Spaceship Flight - This is an example based on the fictional flight plans of a fictional spaceship showing the full monitoring process. Figure 2.3 gives the monitored applications, the specification, the instrumentation and example runs.

The specification represents the property that the spaceship will always land after taking off, will always be on the ground when landing and will never crash into the ground. Formally this can be represented using a fixed point temporal language as

$$\begin{array}{lll} \nu x. & \texttt{takeOff} \rightarrow & \circ \mu y.\ \texttt{land} \wedge onground \rightarrow \circ x, \texttt{land} \wedge \neg onground \rightarrow \circ Fail, \\ & & \neg\texttt{land} \wedge \neg onground \rightarrow \circ y, \neg\texttt{land} \wedge onground \rightarrow \circ Fail \\ & \neg\texttt{takeOff} \rightarrow & \circ x \end{array}$$

The maximal fixpoint, μy, shows that this is a safety property and as such cannot evaluate to true.

The AspectJ instrumentation checks for two different violations. Checking if the property has been falsified on a `land` or `moveV`, and checking that the spaceship has landed at the end of the flight plan by ensuring the monitor is *still_true*. If the specification has been violated the program simply exits, normally it would be useful to print a message detailing what caused the violation. In reactive monitoring the user might choose to alter the flight plan in response to a violation, perhaps by forcing the spaceship to land before terminating.

Two example runs are given. In the first run the result is $false$, as the frontier collapses when z goes below zero. In the second run the result is $still\_false$ as the spaceship is in the air at the end.

```
public class Spaceship{
 public void takeOff();
 public void land();
 public void moveVl(int z);
 public void moveH(int x, int y);
}
```

```
public class Apollo{
 public fly(){
  Spaceship A =
    new Spaceship();
  A.takeOff();
  A.moveV(20);
  A.moveH(100,150);
  A.moveV(-30);
  A.land();
 }
}
```

```
public class Voyager{
 public fly(){
  Spaceship V =
    new Spaceship();
  V.takeOff();
  V.moveV(20);
  V.moveV(-20);
  V.land();
  V.takeOff();
 }
}
```

```
ruler Spaceship {
  observes  takeOff, land, moveV(int);

  state Start : takeOff -> M(0);

  state M(z:int){
    moveV(newZ:int) -> (z+newZ)>=0, M(z+newZ);
    land {: z==0 -> Start; -> Fail; :}
  }

  assert Start, M;
  initials  Start;
  forbidden M;
}

monitor {
  uses S : Spaceship;
  run S .
}
```

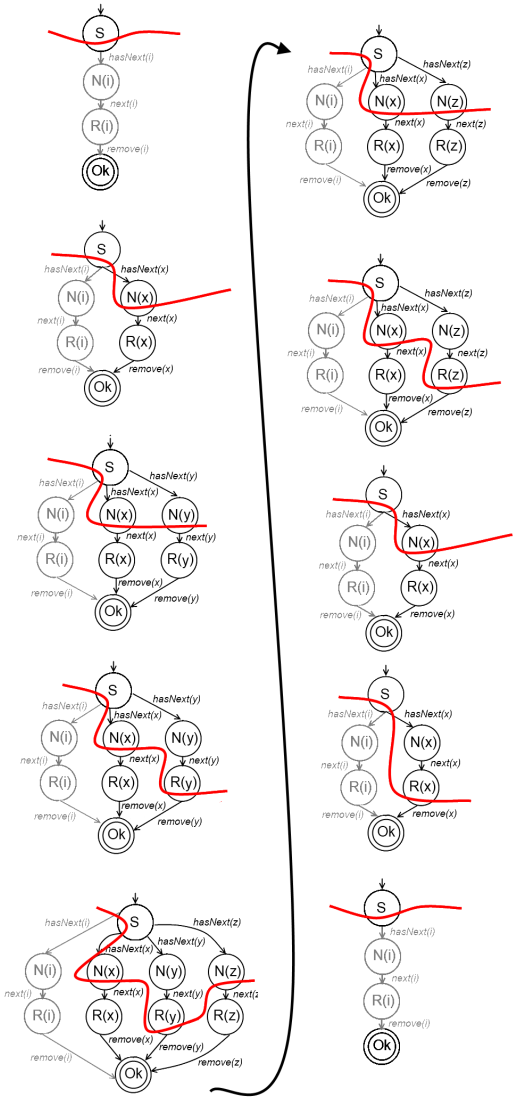
```
public aspect SpaceshipMonitor {
        RuleR monitor = new RuleR("examples/spaceship");

        before() : call(void takeOff()) { monitor.dispatch("takeOff");}

        before(int z) : call(void moveVertical(int)) && args(z){
                if(monitor.dispatch("moveV", new Object[]{z})==Signal.FALSE){System.exit(0);}
        }
        before() : call(void land()){
                if(monitor.dispatch("land")==Signal.FALSE){System.exit(0);}
        }
        after () : execution(void main(String[])){
                Signal result = monitor.dispatchEnd;
                if(result==Signal.FALSE || result == Signal.STILL_FALSE){System.exit(0);}
        }
}
```

Apollo		
Observation	Frontier	Value
	{Start}	
takeOff	{M(0)}	still_false
moveV(20)	{M(20)}	still_false
moveV(-30)	{M(20)}	false

Voyager		
Observation	Frontier	Value
	{Start}	
takeOff	{M(0)}	still_false
moveV(20)	{M(20)}	still_false
moveV(-20)	{M(0)}	still_false
land	{Start}	still_true
takeOff	{M(0)}	still_false

Figure 2.3: Spaceship Flight example.

```
ruler SafeIteratorCheck{

 observes  hasNext(obj), next(obj), remove(obj);

    always Start {
      hasNext(i:obj) -> Next(i);
    }

    state Next(i:obj) {
      next(i) -> Remove(i);
    }

    state Remove(i:obj) {
      remove(i) -> Ok;
    }

    assert Start, Next, Remove;
    initials  Start;
}

monitor {
 uses M: SafeIteratorCheck;
 run M .
}
```



Step	Observation	New Frontier	Value
0		{Start}	
1	hasNext(x)	{Start, Next(x)}	still_true
2	hasNext(y)	{Start, Next(x), Next(y)}	still_true
3	next(y)	{Start, Next(x), Remove(y)}	still_true
4	hasNext(z)	{Start, Next(x), Remove(y), Next(z)}	still_true
5	remove(y)	{Start, Next(x), Next(z)}	still_true
6	next(z)	{Start, Next(x), Remove(z)}	still_true
7	remove(z)	{Start, Next(x)}	still_true
8	next(x)	{Start, Remove(x)}	still_true
9	remove(x)	{Start}	still_true

Figure 2.4: This figure displays an example RuleR specification for checking an iterator performs in the expected way, top left. It also gives a demonstration of how the finite state machine is traversed for a given trace.

Iterators - Figure 2.4 gives a ruler specification definition for verifying if iterators are used properly, this is the `hasNext` property and will be used later. The specification is divided into two parts, the ruler specification and the monitor specification. There are three parts to the ruler specification, a list of observable events, a list of rule definitions, and a list of conditions on rule activations at points in the frontier. The figure also shows how the finite state machine, created from a trace, is traversed by the expanding of the frontier.

2.3.4 Java Implementation

There currently exists a Java implementation of RuleR using AspectJ to instrument applications written in Java. This could also be used to monitor applications written in other languages by reading in log files or communicating with them directly. This implementation is distinct from the language and algorithm, as they could be implemented in many ways in many languages.

AspectJ [25] [6] is an aspect-orientated extension to Java.

AspectJ *weaves advice* into a program at user defined *point cuts* defined in terms of *join points*. A *join point* is a well-defined point in the program and might be a method or constructor invocation or execution, the handling of an exception, field assignment or access, etc. A *point cut* represents a pattern of join points that may match a number of points in the program, and is used to select these points and collect context at these points. The user can then define *advice*, standard Java code, to add before, after or around those points. The *advice* is added to the program when it is compiled using the AspectJ weaver, resultant bytecode contains the advice in-place. For a more in-depth discussion of AspectJ and all its uses see [40].

RuleR Organisation - Figure 2.5 gives an overview of the structure of a RuleR monitoring system, separating the monitored application and the monitor.

The RuleR monitor system receives observations from the instrumented application, which are evaluated as described in section 2.3.2 to get a truth value. It is the user's responsibility to choose what to do with the response.

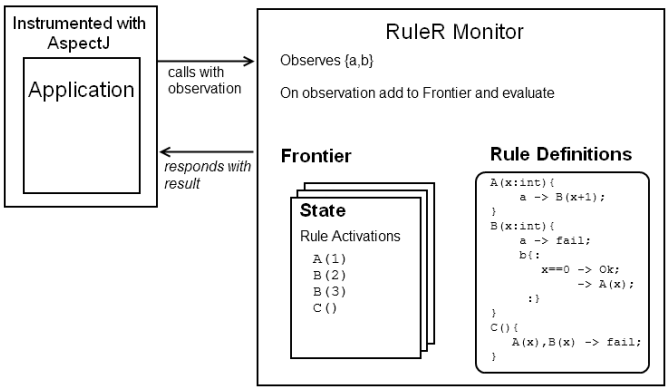


Figure 2.5: Abstract overview of the structure of a `RuleR` monitoring system

Usage - To use the RuleR monitor a RuleR object must be created. The main constructor takes String parameters for input and output files as well as a *timing* boolean flag which when set causes the monitor to add timing events to the execution trace when an event is received. Two `dispatch` methods are provided. The first takes a String and the second takes a String and an array of objects - the first calls the second with an empty array. This represents an event as defined in the specification - to monitor an event `next(obj)` a call of `dispatch("next",new Object[]{iter})` might be made. Each of these methods returns a value from an enumeration containing the five values of the logic used by RuleR, this values is called a signal.

To get the final value of the monitoring process a method called `dispatchEnd` is used. This does not take any parameters and again returns a value from the signal enumeration. This method can only be called once. It also provides a number of statistics on the run, including average step time.

Data Structures - There are a number of important data structures used by the RuleR Java implementation. A brief summary of these is given here to set the scene for later chapters.

Rule System This data structure contains the rules of the system as parsed from the specification. Each rule is given a *rule id* and a rule system is a map from *rule id*s to rule objects.

Rule A rule object consists of a head and a body and give a template for a rule - the RuleCall objects mentioned later represent the rule activations created and used within the monitor. The body consists of an antecedent list and a consequent list of lists - each list made up of term objects.

TermsPair This data structure represents a state the monitor is in and, in version 2 of RuleR, consists of a list of terms representing obligations and a list of terms representing current observations.

Frontier This data structure consists of a list of TermsPair objects.

Term This is the building block of the monitor - arithmetic and logical expressions are terms and are built up of variable and constant terms themselves. Members of the antecedents and consequents of rules are terms, as is an observation or obligation.

RuleCall This is a specific term that represents a rule activation. It consists of a copy of the relevant rule head with parameters replaced with values where relevant. For example the `next` rule from the example given in Figure 2.4 would have the head `Next(i:obj)` and a RuleCall object would consist of a term `Next`(*iter*). When evaluating a rule call the monitor *unifies* the rule call with the relevant rule head and uses the resulting mapping to populate the rule's antecedent to check if it fires and its consequent to generate new obligations.

Rule Monitor This extends a rule system and contains a frontier object. This is the main data structure used by the monitor and includes the `step` that carries out the evaluation of an observation.

2.3.5 LOGSCOPE

A tool based on RuleR called `LOGSCOPE` has been developed and deployed in conjunction with the NASA's Jet Propulsion Laboratory at the California Institute of Technology, as described in [13]. The `LOGSCOPE` tool was developed with the help of the software test engineers working on the next Mars rover mission (MSL - Mars Science Laboratory), and the focus of the tool is to provide a specification language, and accompanying tool, that the engineers can and will use.

The first thing to note about `LOGSCOPE` is that it is an off-line monitor, as it carries out the verification of the program executions after the program has run, this is done through log files. The reasoning behind this approach is that, firstly, most critical programs already have some element of logging and, secondly, that by not adding extra code into the execution of the program the execution of the program will not be effected.

`LOGSCOPE` provides a pattern language and an automaton language, the tool automatically translates patterns from the pattern language into parameterised automaton. The pattern language allows the user to specify the ordering of parameterised events (and their negations), but also allows for unordered collections of events. The automaton language is a subset of the RuleR language one major difference is that `LOGSCOPE` state machine transitions can only be triggered by events, meaning that a rule in `LOGSCOPE` can only depend on a single event. RuleR transitions can be triggered by conjunctions of events, rule activations and their negations. Other differences are that RuleR allows automaton to be parameterised by other automaton and allows monitors to be combined, whereas `LOGSCOPE` does not.

2.4 Summary

This chapter gave an overview of the field of runtime monitoring and presented a number of existing runtime monitoring tools. The RuleR runtime monitoring tool was introduced and its language and algorithm described in detail and a number of examples describing the usage of RuleR were given. Finally a reference was made to the `LOGSCOPE` tool.

Chapter 3

Utilising multicore

This chapter looks at the multicore architecture and the programming techniques that can be used to utilise it. There is a focus on the use of the Java programming language, as this is currently used to implement the RuleR tool. A brief overview of existing parallel tools is given. Finally the experimental machine being used in this project is presented.

3.1 Parallel Computing and Multicore

Parallel computers have been around since the beginning of computing. From the CDC STAR-100 machine built in 1974 which used 10 peripheral processors to complete housekeeping tasks and the 4-processor CRAY-2 build in 1985 to the 44,544 core supercomputer called HECToR currently based in the University of Edinburgh. However this project is not interested in supercomputers, instead the focus is on shared-memory multicore machines.

To clarify things I first discuss two areas of parallel computing I am not concerned with - supercomputers and distributed computing.

Supercomputers are custom built machines for solving large problems. These are now typically all parallel machines, although the first supercomputers were not. Typically these machines are not cache coherent and require specific programming techniques to solve massively parallel problems. This scale of machine is largely out of the scope of this project.

Distributed Computing refers to many computers (I use the term 'computer' as these nodes will typically consist of more than a processor and some memory.) being connected together to solve a computation. These computers can be connected together by standard connection hardware such as Ethernet or high specification interconnects. Again this area of parallel computing is not considered in this project.

In this project SMP and ccNUMA architectures are targeted, both shared memory machines.

Symmetric MultiProcessor (SMP) refers to a multiprocessor or multicore processor with shared memory, coherent caches and uniform memory access. Maintaining uniform memory access is tricky for more than 8 cores. At the moment we see 2 or 4 cores in desktop machines.

ccNUMA refers to a multiprocessor with shared memory, coherent caches and non-uniform memory access. Typically in a SMP processor memory access is kept uniform by having a single main memory module, this obviously is not scalable. Cores my have dedicated memory modules, accessed by other cores by an interconnect. Currently found in the server room, however we might see these machines move to our desks at some point in the future.

It is important to note two things, firstly that with a small number of cores the overheads of parallelisation may overshadow the potential gains and secondly that there has to be sufficient parallelism available to scale to larger numbers of processors. Scalability is an important consideration - more and more cores are being added to chips by manufacturers and a design which cannot make use of more cores to obtain a better result will quickly become obsolete.

3.2 Parallel Programming

The consequence of using shared memory machines is that data structures are shared between threads, this means access is potentially quick and easy but extra synchronisation constructs must be used. This means that the programmer is required to write different programs to take advantage of a multicore machine.

There are two types of parallel algorithms a user can write. In a **blocking** algorithm threads may have to wait for other threads. In a **non-blocking** algorithm threads can always make some forward progress. Consequently a non-blocking algorithm can never experience **deadlock**, where threads are waiting for each other to complete an action before they may continue.

Communication between threads can be **synchronous** or **asynchronous**, in the former case the thread must wait for the communication to complete before continuing, in the latter it does not. Algorithms can also be classified by how often threads communicate - **fine-grained** means that threads communicate often, **coarse-grained** means that threads communicate infrequently and **embarrassingly parallel** means that threads hardly need to communicate at all.

There are four levels at which parallelism can be applied - **task level, data level, instruction level** and **bit level**. Programmers are only concerned with the first two, the former is the decomposition of the task into subtasks that can be executed in parallel and the latter is the decomposition of the data so that the same task can be executed on subsets of the data.

3.2.1 Parallel Java

As RuleR is currently implemented in Java the Java approach to programming a multicore machine is presented[1]. Java provides three basic constructs for writing parallel code.

Threads - Java allows users to create and run Java threads. This is through the `Runnable` interface, implemented by the `Thread` class. A user can choose to implement the interface or extend the class. A user must define a `run()` method, and calling this will cause a new Java thread to be created and the code contained within the `run()` method to be executed, the thread is destroyed when the method returns.

Locks - Each object in Java has a related mutex which acts as a lock. Code running in threads can lock and unlock objects, if a thread attempts to lock an object which has already been locked by another thread it blocks, waiting for that lock to be released. Locks can be gained in two ways - firstly by using a synchronized block of the form `synchronized(`*object*`){`*Body*`}` where the body of the block is executed only whilst the lock of the object is held; secondly by calling a `synchronized` method of an object (or Class object), this method first obtains the lock of the object it is being called upon.

CAS (Compare and Set) - Java provides atomic versions of each of its basic types. These types provide a CompareAndSet method of the form `var.compareAndSet(v1,v2)` which atomically carries out the action `var = (var==v1) ? v2 : var`. This is useful for non-blocking algorithms, for example, one could implement a non-blocking locking system.

The standard Libraries, in particular `java.util.concurrent`, also contain useful features:

Thread Pools - A thread pool is a collection of threads with a queue. Tasks are added to the queue and as a thread completes a task it collects a new task from the queue. Java also provides `Future` objects, which can be used to access the results of tasks sent to a thread pool. Thread pools avoid the costly creation and destruction of threads.

Synchronised Data Structures - The Java library provides a number of synchronized data structures, such as priority queues and hash maps. These data structures guarantee that access to them is synchronised. There is also the option to create a synchronised data structure from an existing one through `Collections.synchronisedCollection`.

Doug Lea's Fork/Join Library as described in [42] offers an additional style of parallel programming. The fork/join parallel programming technique is a parallel version of the divide-and-conquer approach and like divide-and-conquer is often used recursively. The framework developed by Lea is based on the work-stealing methods of Cilk [29]. The framework uses a number of worker threads each with its own work queue - if a worker's queue is empty it will *steal* work from another worker.

[1] There exist alternative approaches to managing threads and synchronisation, for example automated `doall` constructs in Fortran.

3.2.2 Multicore and the JVM

The JVM (Java Virtual Machine), along with the Java compiler, make up the implementation of the Java language. The Java language allows the creation of Java threads, as described above, the JVM dictates how these Java threads are executed. There exist a number of implementations of the JVM, Sun's HotSpot, IBM's J9, Apache Harmony, Jikes and others - all of these map Java threads onto native threads in a similar way.

Sun's HotSpot JVM will be used in this project, which maps Java threads, and their priorities, one-to-one onto native threads (for all current versions, 5 or 6, of HotSpot). As HotSpot runs on Linux, Mac OS X, Windows NT and Solaris it is then down to these operating systems to schedule native threads, and the JVM can only affect this slightly by the way it maps priorities (the user can dictate this).

HotSpot consists of two JIT compilers, a client compiler and a server compiler, and an interpreter. The client is designed to have a fast start up and a small footprint, the server is designed for performance over long periods. Code is interpreted to begin with but as the program progresses hotspots are detected and compiled and the newly compiled code is linked to and ran in its place. Hotspots are detected by associating a counter with methods and loops and incrementing it every time they are run, once this counter overflows the method or loop is compiled. The threshold can be altered by the command line instruction `-XX:CompileThreshold=n` and the defaults are 1,500 for the client JVM and 10,000 for the server JVM.

The HotSpot compiler also offers advanced forms of garbage collection which can make use of multiple cores. A parallel stop-the-world scavenger techniques is used on the young generation, this divides the root set between available threads and uses Parallel-Local Allocation Buffers in survivor spaces to copy reachable objects. Different garbage collection methods can then be used for major collections - `ParallelGC` uses a serial stop-the-world mark-compact method, `ConcMarkSweepGC` uses a mark-sweep method concurrently (a lack of compaction can lead to fragmentation) and `ParalellOldGC` uses a parallel stop-the-world mark-compact method which splits the heap between threads. Additionally to avoid synchronisation costs, giving faster allocation times, each thread is given a Thread-Local Allocation Buffer (TLAB) in the Eden[2] section of the heap.

Lastly HotSpot allows the user to set certain 'ergonomics' dictating how the JVM should manage the heap, which can affect running times considerably. As well as being able to set the maximum heap size the user can specify desired maximum pause times caused by garbage collection and a maximum ratio of normal running time to time spent in garbage collection. The JVM attempts to meet these goals and then attempts to reduce the memory footprint.

[2]This is where all new objects are allocated

3.3 Parallel Tools

There have been previous attempts at utilising parallelism in the area of formal methods. No references to previous attempts at parallelising runtime monitoring tools were found in the literature and the timescale of the project did not allow the manual inspection of publicly available code.

The closest area to runtime monitoring is *model checking*, as previously mentioned this formal method establishes the correctness of a given property by checking each possible behaviour of the system. Due to the previously explained state explosion problem this checking process is not only expensive in time, but also in memory - this gives another motivation for parallelising the process as the use of many machines increases the available memory.

A review of the relevant literature was carried out and it was found that the majority of the previous work focuses on the use of distributed computing rather than shared memory computing - meaning the focus of the design is different as there is a greater concern with reducing communication and shared data.

There are a few examples of shared memory multicore machine focused attempts at parallelising the model checking process.

In [38] a parallel model checker, `PMC`, is presented. `PMC` uses an interesting work stealing approach to balance work among threads - each thread is assigned a private and shared stack and can only add work to their own stacks but can 'steal' work from other thread's shared stacks if they run out of work.

In [37] Holzmann et all present an extension to the SPIN model checker for multicore, shared memory systems. Both liveness and safety properties are supported. Again reasonable results are obtained.

In [10] a parallel LTL model checker is developed. Barnat et al. take an algorithm that uses weak Büchi automata to represent LTL properties and apply a number of parallelising techniques to it. They take a message-passing approach using per thread FIFO queues protected by locks to communicate messages. Their work focuses on how a previously distributed approach can be adapted for a shared memory machine, considering different communication and memory allocation effects.

Although the above previous work represents successful utilisation of multicore systems for solving the model checking problem the workload present in the model checking process is significantly different from that present in runtime monitoring. In model checking the entire state space is considered and checked in one process. The runtime aspect of runtime monitoring means that the process consists of many small, dependent, time-sensitive, processes.

3.4 Experimental Machine

The experimental machine used in this project is an Apple Mac Pro. The machine has two 2.26GHz quad-core Intel Xeon processors, with 8MB fully shared L3 cache per processor, and 16GB of SDRAM.

The Nehalem microarchitecture, the family of processors the Xeon chip belongs to, is presented in Figure 3.1. The microarchitecture is superscalar and therefore 4 instructions can be executed in a single cycle. The microarchitecture has 32KB instruction and data L1 caches, a 256KB L2 cache and a 8MB shared L3 cache. The L3 cache is inclusive, meaning that it contains all entries in L1 and L2 caches.

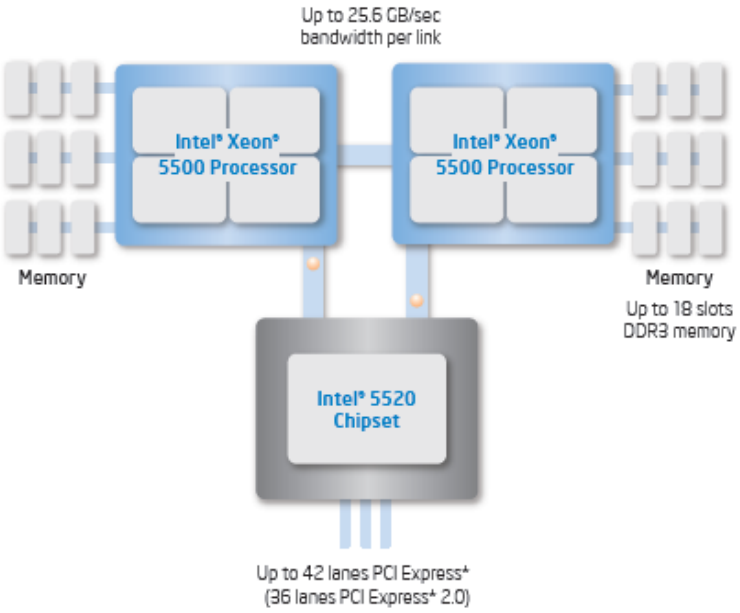


Figure 3.1: The Intel Xeon processor, a Nehalem architecture.

The technologies contained within this chip are

TurboBoost - This technology, when activated, dynamically alters the frequency of the processor to optimise performance based on power consumption and processor temperature. The technology can be activated by the operating system.

Hyperthreading - This is the most important technology. Intel's hyperthreading technology is a form of simultaneous multithreading, where multiple threads of execution are executed simultaneously. In hyperthreading two threads are executed on one core simultaneously, taking it in turn to execute an instruction unless the other thread is busy. This reduces computation latency, preventing cycles being wasted during I/O and other cycle-heavy operations. It may give the appearance of zero-cost context switching, as the information from two threads is held on chip and can be switched between instantaneously.

QuickPath - This is an interconnect technology providing point-to-point connectivity between processors and each other and the I/O hub. QuickPath provides a maximum connection of 25.6GB/s.

This information was sourced from the Nehalem whitepaper[3].

To summarise the two quad-core Xeon processors provide 8 cores with 16 hardware threads due to hyperthreading. With more than one core the microarchitecture provides cache coherent non-uniform memory access - each core has its own memory module, in this case with 8GB of memory in each. The QuickPath interconnect technology provides a very fast but non-uniform access to memory modules of other cores.

[3]found on the Intel website at `http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf`.

3.5 Summary

This chapter discussed how multicore systems fit into the area of parallel computing and gave a short overview of things to consider when programming a parallel machine. A brief overview of other parallel tools was given. Finally the experimental machine was presented.

Chapter 4

Analysis

The previous two chapters introduced the field of runtime monitoring, including an overview of the RuleR tool, and explored the current field of multicore machines, the associated architectures and the software designed for them. This chapter presents an analysis of the performance and interference overheads introduced by the Java implementation of the RuleR tool and then considers possible approaches to tackling these overheads.

From this point 'RuleR' or 'RuleR tool' refers to the Java implementation only, unless specified otherwise.

4.1 The RuleR Tool and its Overheads

Section 2.3 gave an overview of the RuleR runtime verification tool, this section outlines the overheads introduced by the Java implementation of this tool.

Recall that the monitoring process consists of instrumentation generating an execution trace to be processed by the monitor. It is important to note that instrumentation and monitoring are separate activities. It does not matter to the RuleR monitor how the execution trace it processes is generated.

There are a number of ways a program could be instrumented to generate the execution trace - The program could be instrumented directly by hand to pass observations to the monitor, or observations could be saved to file at runtime and then the monitor run offline (after the monitored program has ran) afterwards by reloading and feeding the observations to the monitor. In the typical case, and the usage this project focuses on, the monitored application is instrumented using AspectJ, creating an instance of a RuleR monitor and dispatching events to it.

As the instrumentation and monitor are separate the overheads they may introduce are independent and will be considered separately.

4.1.1 Instrumentation

The purpose of instrumenting a program is to generate an execution trace to be monitored. To do this relevant points in the program are selected and code is inserted at these points to communicate that the

point has been reached to the monitor. An event may require information about the current state - it is the job of the instrumentation to collect this and pass it to the monitor. Whether this process is done manually, by writing in the instrumentation code by hand, or automatically, for example by using an AOP (Aspect Oriented Programming) tool such as AspectJ, this additional code contributes an overhead.

AspectJ *weaves* the specified instrumentation code into a program, having the same effect as manually identifying the relevant points and typing the code in by hand. There is some additional cost involved in creating and instantiating an `aspect` object per aspect.

Depending whether events are being saved to file or sent to a monitor there will be some setup costs in addition to the creation of the aspect. In the case considered here this consists of the initialisation of the RuleR monitor by parsing the specification file. This is a one off cost and is dependent on the size of the specification. When there are many calls to the monitor this overhead will not be noticeable (being amortized over all calls) but if there are very few calls it may become significant. However in terms of interference the monitor initialisation and aspect creation will cause a long pause in the monitored application.

At every event location (instrumented point in the program) a reasonably constant amount of work will be required for instrumentation. This consists of collecting the relevant information about the current state of the program, constructing an event and deciding whether to monitor that event - perhaps through checking boolean flags or keeping track of the previous event.

The instrumentation overhead should not be large and will typically remain constant irregardless of the property being monitored - it is expected that this overhead will be much less than the monitoring overhead.

4.1.2 Monitor

The overhead introduced by the monitor is the amount of time taken to process an event multiplied by the number of events. However the time to process an event is not constant and depends on a number of factors.

The RuleR monitor receives an observation and returns a signal. Each observation has a name and an optional array of objects representing the parameters to that observation. Importantly each object passed as a parameter in an observation is only referred to by a `WeakReference`, this prevents memory leaks and allows the monitor to remove terms referring to objects which have gone out of scope. The shallow size of an observation is 80 bytes - meaning just over 400 rule activations can fit into L1 cache on the experimental machine.

The table below gives an outline of the steps the RuleR tool goes through for each event with a rough complexity for each step. Given the number of states s, the number of rule activations r, the number of rules fired f and the overall number of steps i. Typical values for these would be 1 or 2 for s as non deterministic properties are uncommon, again 1 or 2 might be a typical value for r but here we are more interested in properties that generated hundreds of rule activations, typically only a handful of rules will fire on a single step, and a monitor may be ran for only a few steps or a few million steps depending on the monitored application.

		Complexity
1	Check if the observation is the end of the input. If it is ensure that no forbidden rules exist in the frontier.	$O(1) + O(r/i)$
2	Check that the frontier is not empty	$O(1)$
3	Check that the frontier still contains obligations	$O(s)$
4	Merge the observation with the frontier, by adding it to each state. Evaluate the state with respect to the observation and check that the state has not collapsed.	$O(s) + O(r)$
5	Create a new frontier object by evaluating each rule activation in each state in the frontier.	$O(r)$
6	If the specification specified some `assert` rules, check that at least one has fired, if not return false and halt monitoring.	$O(f)$
8	If the specification specified some `success` rules check if one has fired, if so return true and halt monitoring	$O(f)$
9	If the specification specified some `forbidden` rules check if the frontier contains one, if so return $still\_false$ otherwise return still_true	$O(r)$

The time to complete a step is in the order of microseconds - during initial experimentation timings were seen varying between one and a few thousand microseconds. For the `hasNext` workload used frequently in Chapter 6 average times of between 1 and 10 microseconds were seen.

Working on the assumption that s and f are small it can be seen that that the work required to process an event is heavily dependent on the size and structure of the frontier. In version two of RuleR the frontier was implemented as a list of states where each state was a list of rule activations, in a later version a map of lists is used to represent a state to separate rule activations for particular rules.

The size of the frontier, r, is determined by a combination of the specification and the received execution trace so far. Additionally the number of times the frontier must be searched as part of checking if a rule activation will fire is dependent on the definition of the rule in the specification. Therefore much care should be taken in writing the specification to avoid the unnecessary monitoring of events. Additionally if one rule is used a large amount an extraneous condition can add a large overhead when summed up over all events.

For example the rule

```
state Ready(o:obj){ A(o), B(o), C(o) -> Go(o);}
```

is dependent on three other rules requiring three searches of the current state. In the list implementation this involves a linear search of the state and in the map implementation this involves jumping to the list of rule activations for the given rule and then a linear search of that list.

Another cause of large frontier sizes is *garbage terms* - terms referring to objects which have gone out of scope in the monitored application and so can never be fired. As previously mentioned the use of weak

references in RuleR allows for easy detection of these. However a mechanism must be in place for removing these terms. In version 2 of the tool rule activations are checked whilst building the new frontier and the garbage terms are ignored, not being added back in. This can add considerable unnecessary overhead when the majority of terms are not garbage - especially if the terms in the frontier are complex as the garbage check is recursive.

4.1.3 Possible Consequences of these Overheads

The consequences of these overheads go beyond increasing the execution time of the monitored program as they may **interfere** with the execution of the monitored program. This may increase the execution time beyond just the overheads or may alter the behaviour of the program. The consequences will depend on the original behaviour of the monitored program.

In a *multithreaded program* increasing the time for a single operation to complete may cause operations to be ordered differently, if these operations are not independent this could lead to different behaviour. The likelihood of missed wake-ups (if they potentially exist) may increase. However in a multithreaded application it may be possible to see lower than expected overheads as better load balance and waiting times may hide some of the overhead and some operation re-orderings may be more efficient.

Additionally as the internals of RuleR are not currently thread-safe the dispatch method must be synchronized, causing a serialisation of threads in the multithreaded program vastly increasing the impact of monitoring.

If the program *interacts* with users or external programs *responsiveness* may be effected. Also the correct behaviour of *time-sensitive* systems, such as machine control systems, is dependent on certain operations executing within time limits. Additional overheads will reduce the variation allowed in these operations. Also if the intention is to monitor these operations to verify they execute within the given time limits the overheads place a restriction on the monitorable limits. For example if it takes 100 microseconds to evaluate an event it is not possible to ensure that two events happen within 50 microseconds of each other.

Another side-effect of the monitoring process is the use of memory. If the monitored application is memory intensive the use of extra memory by the monitor could lead to additional garbage collection, increase in heap sizes, paging, segmentation of data in memory and possibly running out of memory. Additionally the monitoring process may become a *memory-leak* if the monitor maintains references to objects used by the program after they should have gone out of scope, a situation avoided by RuleR but one to be aware of if changes are to be made.

4.2 Tackling the Overheads

The previous section discussed what overheads exist and this section considers how to tackle those overheads. As previously shown the processing of each event contributes an overhead. Therefore the two approaches are to reduce the number of processed events and to reduce the amount of work required to process an event. This section focuses on a number of ways the amount of time taken to process an event

can be decreased inside the RuleR monitor and ways to decrease both the amount of time to process an event, and the number of events processed, outside of the monitor, from the perspective of the monitored application. The focus is on the use of concurrency in tackling overheads.

4.2.1 Inside RuleR

The first thing to attempt is the parallelisation of the processing of an event. The most work intensive part of this process, and the part which scales most with problem size, is the evaluation of the frontier. This can also be split up into small, largely independent, parts as the evaluation of a rule activation depends only on the previous frontier and the current observation.

Another possible form of parallel activity would be to factor out certain maintenance tasks to be run in parallel with the monitoring process. There are two tasks that can be factored out:

1. Clearing garbage terms - If this were done in parallel, ideally between calls to the monitor, then the additional overhead of checking each term can be removed.

2. Maintaining states as sets of rule activations - When adding a rule activation to a state a containment check is carried out, this check can become very expensive for large states. A possible solution is to remove redundant rule activations (after they have been included) in parallel for states over a certain size.

There also exist a few optimisations possible which do not directly employ concurrency.

- Early return and early failure - Firstly when each rule activation is being checked to see if it fires this check should detect failure as soon as possible to avoid doing unnecessary work. Secondly if the specification is going to fail it should fail as soon as possible, although as this is not a common case its optimisation is not very important. Thirdly a result should be returned as soon as possible. One possible approach to this is to factor out the work to process an event into work essential to generating a result and tidy up work to be performed afterwards. These could then be presented as separate methods called alternately - this would not reduce the overall amount of work but would mean that a result was returned as quickly as possible and a monitor running in parallel could carry out tidy up work without whilst the monitored program continued running.

- A persistent frontier - Currently the frontier object is reconstructed on every step, meaning that a new object must be created and allocated on the heap and all the persisting rule activations copied across - this is particularly wasteful for `always` rules. An alternative solution is to organise the frontier in such a way that it remains persistent and rule activations are removed and added as required.

- Restructuring the frontier - The frontier is accessed at two points - it is iterated over to check and fire the rule activations contained within it and it is searched when a rule appears in the condition of another rule. The first use requires cheap iteration over the structure, the second use suggests a need to quickly jump directly to a rule. The former usage is far more prevalent than the latter and should therefore be the focus.

4.2.2 Outside RuleR

When considering what can be done outside of RuleR there are two focuses - firstly looking at how the number of events processed by the monitor can be reduced and secondly considering how the perceived time to process an event can be reduced.

Reducing the Number of Events Processed - This requires an understanding of the property being processed and the events required to properly check that property. The first place to reduce the number of events required is in the specification - even though a number of events may be involved in the process being monitored only the ones required to check the property should be included. The second place to reduce the number events required is in the instrumentation - with many specifications it is possible through one or two simple boolean checks to filter out a lot of events. For example if one kind of event is only of interest once another kind of event has occurred then a boolean flag could be used to signify and check this. Whilst this will add an additional cost to instrumentation the trade-off will be worthwhile if the number of events processed is significantly reduced.

This assumes that as soon as an event reaches the monitor it will be processed. But if this assumption were relaxed and the monitor were allowed to choose not to monitor an event then it could analyse the specification to generate a number of quick checks to perform on incoming events. This is not implemented in this project but is discussed in further detail in section 8.3

Reducing the Perceived Time to Process an Event - Doing this beyond reducing the actual time to process an event requires carrying out some of the processing in parallel with the monitored application. To do this the RuleR monitor must be run in its own thread. However given the present organisation of the monitor this will merely introduce a synchronisation overhead because on every event the monitored application waits to discover the result.

Therefore the next step is to reduce the number of these synchronisation points where the monitored application waits for the monitor to deliver a result, let the events leading to these synchronisation points be referred to as *significant events*.

These significant events must be chosen by the implementor of the specification and instrumentation. A significant event may represent a point where the specification is likely to fail or where some action must be taken - often it can easily be concluded from inspection of the specification which events could never cause the monitor to fail. Again this represents an opportunity for the monitor to use the specification to decide how it deals with events.

The work of processing all the events between significant events can then be covered by the monitored program. Therefore the organisation of events and significant events in the execution trace will have a large impact on performance and interference.

There are a number of relevant organisations of the execution trace, as shown in Figure 4.1.

1. **All events are significant** means that the monitor and monitored application must synchronise on every event. It is not possible to use parallel execution to cover up the overhead of monitoring

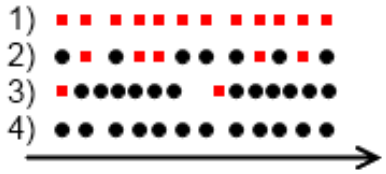


Figure 4.1: An overview of different execution trace organisations. 1) All significant events, 2) Evenly spread events, 3) Clustered events and 4) No significant events. Normal events are black circles. Significant events are red squares.

unless the reply can be delayed by a number of events - this would not work if an action had to be taken.

2. **Evenly spread events** with significant events also evenly spread. This may be the case where a frequently used object is monitored and some operations are important enough to check - for example when pushing an object on to a stack there is no need to wait for a reply as this action could not break the property that a pop cannot be called on an empty stack, but when popping an item the property could be violated if the stack is empty, so it could be worth waiting. The amount of monitoring overhead hidden will depend on the timings between events and the number of events between significant events.

3. **Clustered events** with significant events appearing at the beginning of the cluster. This may be the case when an object is being monitored intensively for a short period and it is necessary to check that all is well before beginning monitoring. If the time between clusters is long enough then almost all of the monitoring overhead can be hidden.

4. **No significant events** means that events can just be dispatched without waiting for a reply, with the result being checked at the end of the monitored application. This should reduce interference in terms of pause times but as objects must be kept alive to monitor other memory related interference could occur.

An alternative approach is to decompose the execution trace between different monitors running in their own thread. However events which are dependent on each other must be evaluated by the same monitor, so some way to divide the execution trace into a number of independent groups is required. The first obvious division is by thread - if the property is per-thread then each thread (or group of threads) in a multithreaded program is allocated a different monitor, this also reduces the serialisation of those threads.

Alternatively a specification specific tag could be used - if the specification monitors the use of an object and events related to a particular instance of that object are independent of events related to other instances of that objects then the object instance's hash code could be used as a way to divide the execution trace. Figure 4.2 demonstrates this. Although care should be taken when using an objects hash code in this way - it is not guaranteed to stay the same throughout the execution of the program (for example Collections combine the hash codes of their contents), and can be overridden - knowledge of the object being monitored should be used to select an appropriate tag.

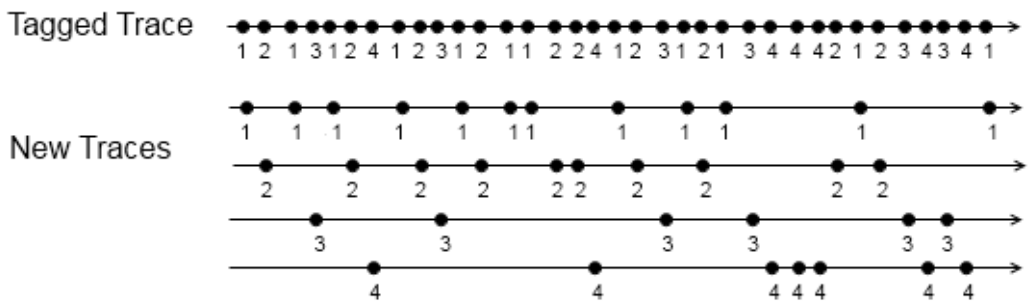
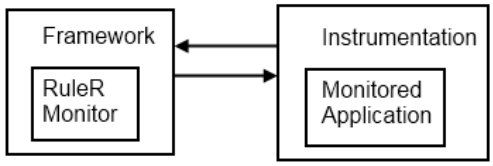


Figure 4.2: The tagged approach - by tagging events in the execution trace a number of smaller traces can be produced with more time between events.

4.2.3 A Design

This analysis leads to the conclusion that there are two avenues to pursue - firstly to extend the RuleR monitor to include the optimisations described above, and secondly to implement a framework to wrap the monitor in to allow appropriate filtering and buffering of events.



The first aim of this project is to explore ways in which multicore can be used to increase the performance and decrease the interference of the monitoring process. This design addresses these in the following ways

Increase Performance - Firstly basic optimisations will reduce the amount of time required to process each event. Evaluating the frontier, as well as carrying out maintenance tasks, in parallel will also decrease the amount of time spent on each event. This makes it more likely that on a significant event only that event will be evaluated.

Decrease Interference - The monitoring of non significant events will only require the dispatch of the event. There is the possibility that the evaluation of significant events will take a lot more time than a single event would previously. This will depend on the monitored application and placing of significant events.

In terms of scalability the two approaches which can scale are the parallel evaluation of the frontier and the notion of decomposing the execution trace through tagging. Otherwise these approaches do not allow the monitoring process to run faster with more threads.

4.3 Reflecting on the Monitoring Process in General

This analysis of overheads, and plan of how to tackle them, can, to an extent, be generalised to the monitoring process in general. Specifically the discussion of ways to reduce the overheads caused by synchronisation between monitor and monitored application. If a monitor can be abstracted as a black box which given an event returns a signal the above discussions focused on what occurs outside the monitor still apply.

There exists a separation between the generation and communication of the execution trace by the instrumentation and the processing of events by the monitor. By running the monitor in parallel it will always be possible to hide some of the overhead by separating out significant events. In general constructing the problem to reduce the size and complexity of the execution trace will increase performance.

The specification can be seen as a piece of data that can be tackled using a data parallel approach - in essence this is what the taggable approach is doing. As all monitors act over an execution trace this way of decomposing the execution trace can easily be used without altering the monitor at all.

4.4 Summary

This chapter looked at the overheads introduced by the monitoring process, including both those that are introduced by instrumentation and those introduced by the evaluation of the event itself. Possible approaches to tackling these overheads were discussed, focusing on the use of concurrency, which led to a design for a framework. The next chapter describes how the framework was implemented.

Chapter 5

Developing an Experimental Framework

The previous chapter gave an analysis of the overheads introduced by the RuleR tool and outlined possible ways of tackling them. This chapter presents the experimental framework developed to explore and evaluate these different approaches.

5.1 The Framework

The experimental framework consists of a number of components sitting in-between the monitor and the instrumented application. This framework is designed in a modular way so that it can be easily extended and used with other instrumentation techniques or monitor tools.

5.1.1 Components

The framework consists of five components, as presented in Figure 5.1. One or more implementations have been developed for each component for this project - section 8.3 describes other possible interesting implementations that time constraints did not allow. The components are:

- **Reply Mechanism** This component is in charge of how the instrumented application communicates with the framework, and most importantly how replies are dealt with. A reply mechanism object is created by the instrumented application, which then uses `dispatch` and `dispatchEnd` methods provided by the reply mechanism to communicate with the monitor. These dispatch methods may

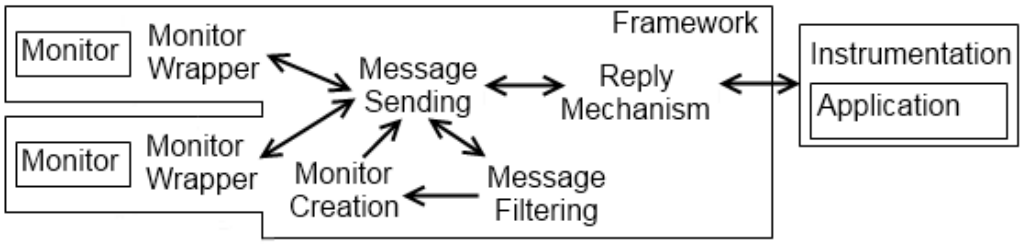


Figure 5.1: The components of the framework fit together to provide communication and event filtering in between the monitor and instrumented application.

or may not return a signal depending on their implementation. The reply mechanism object creates and uses a message sending object. A message object is created for every dispatch call and sent using the message sending object. The different reply mechanism implementations are outlined below in section 5.1.2.

- **Message Sending** - This component is in charge of communicating and filtering messages and uses a message filtering object and a monitor creation object to carry out those tasks. There are single monitor and multiple monitor implementations of this component. The implementations directly call the dispatch method of the monitor or the reply method of the reply mechanism.

- **Message Filtering** - This component filters messages and may remove or alter a message. The message filtering object makes use of a monitor creation object to create monitors. There exists a basic implementation of this component and two extensions - Latest and Halting. The base implementation deals with monitor creation and completion and per-thread messages (see section 5.2) where relevant. Latest filtering returns the most recent signal. Halting filtering halts the application and monitor immediately on detecting a `false` signal.

- **Monitor Creation** - This component is used to create monitors. It is a singleton object and can be used to manage resources between existing monitors. Only one implementation of this component exists - this uses an `options` object giving parameters for the monitor constructed by the instrumentation to create a monitor.

  The options available include a timing flag, a per-thread option, a parallel version option and a number of threads. This is in addition to giving specification and output file locations.

- **Monitor Wrapper** - The monitor used, in this case RuleR, is encapsulated in a monitor wrapper object. Two implementations exist - the first caries out a direct call on the monitor object it is wrapping and the second uses a message buffer which it evaluates by running the monitor in a separate thread. For each message the monitor wrapper uses the monitor to evaluate the message and then tags it with the signal returned by the monitor. The message is then sent back to the reply mechanism object via the message sending object, where it will be filtered again. Different monitors could technically be used here, but this project is only concerned with RuleR.

This means the monitoring of a single event involves the following

1. A dispatch from the instrumentation to the reply mechanism creates a message object out of the information supplied about the event.

2. The reply mechanism calls the send method on its message sending object with the message. This filters the message - if it is a special message to do with monitor creation or completion it is removed and this is dealt with separately, if it is a per-thread message this is expanded (as outlined in 5.2).

3. The message sending object calls the dispatch method on the monitor wrapper, this will either immediately evaluate the message or add it to a buffer.

4. Once evaluated the message is tagged with the response and returned to the reply mechanism via the message sending and message filtering objects. Halting filtering will check for `false` replies and latest filtering will maintain a record of the latest response.

5. The message sending object will call reply on the reply mechanism object with the message. This will either return the response directly to a blocked application or will store it for retrieval later.

Originally it was thought that the use of weak references by the RuleR monitor to utilise Java's garbage collection to remove garbage terms would cause problems when monitoring of an event took place after that event had occurred. The problem would be that objects referred to by the event would go out of scope and therefore be collected before they were monitored. Therefore a mechanism was implemented to keep objects alive, adding in garbage collection events into the execution trace.

However the fact that the message objects refer to these objects with a strong reference means that the objects can not go out of scope until that message has been cleared by the reply mechanism object. This may cause garbage terms to last longer in the RuleR monitor if messages are not retrieved by the monitored application frequently.

The different implementations, along with specific reply mechanism implementations, are combined to build *architectures* designed to be specifically suited to dealing with a particular behaviour of monitoring as described in the previous chapter.

5.1.2 Architectures

Each architecture uses an individual implementation of the reply mechanism component. The choice of other components may be fixed, for example a direct call monitor wrapper can only be used with a reply mechanism which blocks. Each architecture can be ran in per-thread mode as explained in the next section. The architectures are:

BlockingDirectCall This represents the normal operation of the RuleR tool. The monitor does not run in its own thread, instead it is directly called by the reply mechanism via the message sending object. There is the option to use halting message filtering, but latest message filtering would have no effect. It would always be better to use the monitor on its own rather than within this architecture but this was implemented for comparison purposes.

BlockingPrevious This relates to the behaviour where every event is a significant event but there is some reasonable time between them and the response from monitoring each event can be delayed for a short time. The monitor is run in parallel and every dispatch returns the result of the previous dispatch. Again only the halting message filtering would be of benefit.

SendThenWait This relates to the behaviour where events are clustered and there is one significant event at the beginning of each cluster, however it could be used in other instances. The architecture offers two dispatch methods - a blocking and a non-blocking one. The non-blocking dispatch returns

straight away after sending the event and increasing a counter. Whenever a reply is received the counter is decremented. The blocking dispatch sends the event, increases the counter and then waits for the counter to reach zero before returning the most recent reply. In the per-thread mode these counters are per-thread.

Both the latest and halting message filtering extensions can be used here. If latest filtering is used then the non-blocking dispatch will return the latest result, which may mean the same result is returned more than once if a new result has not been received, otherwise the dispatch returns null.

NonBlocking This relates to the behaviour when there are no significant events. The `dispatch` method sends the event to the monitor and returns straight away and, unless latest message filtering is turned on, returns null. The `dispatchEnd` method blocks, waiting for all events to be processed. Again both the latest and halting message filtering extensions may be used.

TaggedNonBlocking This relates to the behaviour where the execution trace can be divided into independent sub traces involving events identifiable by some unique value. The dispatch method used by the instrumentation for the architecture includes a tag parameter which is used to spread events between a number of separate monitors.

On creation a number of groups is specified and then for each group a monitor running in its own thread is created. The tag given in the dispatch call is passed through a parameterised hash function to give an index to a group. Again both halting and latest message filtering can be used. A call to `dispatchEnd` sends an end event to all monitors and collates the result.

PostNonBlocking This relates to the behaviour where there are no significant events and reducing interference is more important than increasing performance. Events are dispatched in the same way as in NonBlocking but the monitor threads are not started until `dispatchEnd` is called. This means that the running monitor threads cannot interfere with the running application although there is, of course, still some overhead. It makes no sense to use the halting and latest message filtering extensions.

This architecture could also be used to monitor a backup property - if a property could be split into an easily monitored less strict version and a stricter yet harder to monitor version then the less strict version could be monitored concurrently with the application and if this fails the stricter version could be ran afterwards. If failure is not the common case this should lead to better performance as well as reduced interference.

5.2 Per-thread Properties

When monitoring a multithreaded application some properties can be monitored independently for different threads, for example checking the usage of a thread local data structure, and some properties require a serialisation of the execution trace to monitor, for example the use of shared data structures and locking.

5.2.1 The Problem of Serialisation

If a single monitor object is used to monitor a per-thread property then the threads of the monitored application will serialise on the monitor - one thread will have to wait for a different thread to finish using the monitor before it may use the monitor and continue. This may introduce large, unnecessary, overheads.

Figure 5.2 demonstrates the effects of serialisation on a simple multithreaded workload (taken from the microbenchmark described in section 6.7). A two threaded example saw a two times slowdown and a sixty-four threaded example ran an order of magnitude slower.

To avoid this serialisation dispatch methods in the reply mechanism are not synchronized. The reply mechanism object keeps a synchronised map of reply queues, using the thread's *thread id* as a key. Messages are then tagged with a *thread id* by the reply mechanism object so that they can be added to the appropriate reply queue when returned.

For the evaluation of these per-thread messages two solutions were implemented, as described in the following two sections.

5.2.2 Outside the Monitor Wrapper

A separate monitor and monitor wrapper are created and run for each monitored thread. No monitor is created when the framework is initialised, instead an abstract *monitor id* is generated and returned and then used by the instrumented application. Messages sent to this abstract *monitor id* are intercepted by the message filtering object which uses a combination of the abstract monitor id and thread id to lookup the relevant concrete monitor wrapper and replaces the destination monitor in the message. If a concrete monitor wrapper does not exist it will be created through the monitor creation object.

When a dispatch end message is sent to the abstract *monitor id* this is replicated and sent to all monitors related to that *monitor id* and the results are collated and added to the original message to be returned to the reply mechanism.

Once a thread ends the monitor evaluating events from that thread can stop running. To enforce

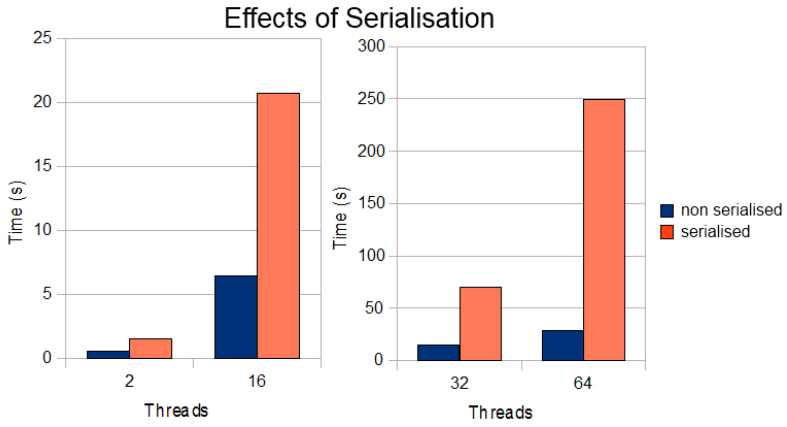


Figure 5.2: The effects of serialisation

this a helper thread runs alongside the monitor threads to detect when their associated monitored thread ends. On detecting the end of a thread this helper thread sends a 'complete' event to the monitor, which calculates the final result for its execution trace and ends any additional threads it may be running. The helper thread detects the end of a thread by storing a reference to the monitored thread at monitor creation and checking if the thread is still alive at regular intervals.

This approach means that a monitor is created for each thread. If each thread only produces a small number of events but there are a large number of threads this will lead to large costs in terms of thread creation and thread scheduling. The `TaggedNonBlocking` approach presents an alternative as groups of threads can be joined together.

5.2.3 Inside the Monitor Wrapper

The problems with the outside monitor wrapper solution are that many threads might be created, the specification must be parsed for every monitor created and translating the abstract *monitor id* to the concrete *monitor id* introduces an overhead. The solution inside the (buffered) monitor wrapper is to use a number of different buffers, each with its own thread all operating on a single monitor object. This requires the monitor to have a per-thread option - this is added to the RuleR monitor as explained in the next section.

A number of different strategies can be selected here - all events could be monitored with a single thread, a fixed number of threads could be used or a thread per thread could be used. As in the above the ends of these monitored threads must be detected to terminate the corresponding monitor thread.

5.3 An Optimised Multicore RuleR Monitor

The previous chapter outlined possible optimisations to the RuleR monitor. This section firstly describes initial optimisations made to the serial version of the tool, then presents the cleaner thread which carries out maintenance tasks, and finishes with an outline of the different parallel implementations of the tool that were developed. Figure 5.3 shows how the different versions developed relate to each other.

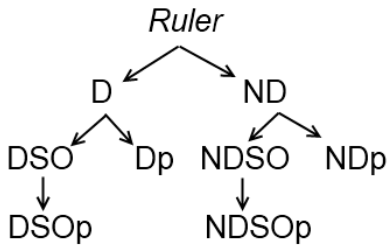


Figure 5.3: The eight different versions of the RuleR monitor are D - Deterministic, ND - Non Deterministic, DSO - Deterministic Single Observation, NDSO - Non Deterministic Single Observation - and the parallel versions of each of these.

5.3.1 Initial Optimisations

Through a combination of experimentation and profiling the following optimisations to the serial version of RuleR were identified and implemented.

Adding an Internal Per-Thread Option - As explained previously for the per-thread approach within the monitor wrapper to work a per-thread option must be added to the RuleR monitor.

A per-thread flag is added to RuleR's constructor. On construction the specification file is parsed as normal, this can then be shared by different monitors as it will not change during the execution of the monitors. When the flag is set to true each dispatch must also be tagged with a thread id. The first time a *thread id* is seen a new Rule Monitor object is constructed and stored in a map, referenced by the *thread id* - all dispatches including that *thread id* will then lookup and use this object. The only point of synchronisation is the usage of the map.

Making Deterministic and Single Observation Versions - It was noticed that the majority of specifications encountered were deterministic and only dispatched a single observation on each step. In these cases it is unnecessary to use collection data structures where the collections will only ever contain one element.

The two data structures relevant are the frontier and the specific term data-structure used to hold observations. Both are implemented as lists - the frontier as a list of states and the observation term as a list of terms. Four versions (as shown in Figure 5.3) were developed -

1. Deterministic (D) - this uses a single state to contain active rules and can hold multiple observations.
2. Deterministic Singe Observation (DSO) - the same as D but can hold only one observation.
3. Non-Deterministic (ND) - a number of states can contain the active rules and are stored in a frontier object, multiple observations can be held.
4. Non-Deterministic Single Observation (NDSO) - the same as ND but can only hold one observation

The choice of which Rule Monitor object to create is decided when parsing the specification. In the current organisation of RuleR the only instance where two observations can be evaluated on a single step is if timing is activated - then a timing observation is added at the dispatch point. Therefore if the timing flag is not selected in the constructor an SO Rule Monitor is created. The determinism of the specification is detected after parsing the rule system - rules with parallel consequents or a parallel initial state indicate non-determinism.

Splitting each State - There are two times a state object is accessed - it is iterated over to check and fire the rule activations it contains and if a rule activation depends on another rule it is searched to check if that rule is present. There will therefore always be a trade-off between focusing the data structure on either of these operations.

The state contains a list of rule activations, this is split into four parts - one for each of the rule types (always, state and step) and one for observation obligations. This means that when iterating over the state the lists must be combined and when searching for a rule activation only rules with the relevant rule type are checked. A hidden advantage is that it becomes easy to remove only non-persistent rules - this is helpful in making states persistent

This implementation represents a compromise between the previous version using a list and the alternative version using a map - administration costs with a map object are larger and as the common usage is to iterate over the frontier these can be costly. Furthermore the advantages of using a map object are limited as in the majority of cases only a small number of different kinds of rules are present in a specification and there are often many instances of the same rule.

Making the States Persistent - To avoid the cost of constructing a new state object and copying the persistent rules every time a state is evaluated the state objects are made persistent. On each step all non-persistent rules are cleared from the state and the `state` rules which have not fired are added back in along with the new obligations. So that rule activations are still evaluated independently no changes are made to the state object until the end of evaluating the state - new additions are stored in a buffer and flushed through at the end. This flushing process also deals with rule negations.

Removing BindingException and `foreach` - A binding exception was a subtype of the standard exception used in the unification process - if two terms did not unify then a binding exception was thrown and caught later on. The unification process is used throughout the monitoring process but mainly when checking to see if a rule activation will fire as the rule head and rule activation must be unified and the observation may have to be unified with terms within a rule's antecedent.

Early on it was found that the use of binding exceptions and `foreach` loops were very expensive and alternative implementations were used. In the binding exception case `null` was returned and checked for instead. In the `foreach` case collections were iterated over directly where possible.

5.3.2 The Cleaner Thread

Once a state reaches a certain size the operations of removing garbage terms and checking for redundant terms when adding to the state become expensive. To remove garbage terms `isGarbage` must be called on each item in the state and the current method for ensuring terms are not repeated in the state is a call to `contains` when the term is added.

To combat these overheads the cleaner thread operates in parallel with the monitor carrying out these operations. All garbage is collected by the cleaner thread and when a state reaches a certain size it stops checking for redundant rules and these are removed by the cleaner thread.

The cleaner thread and monitor must synchronise in some way to allow the state to be updated safely and it is important to minimise the interference with the monitor. However it is also important to ensure that the cleaning is done as redundant terms can quickly multiply and a state inflated with garbage and redundant terms takes longer to iterate over.

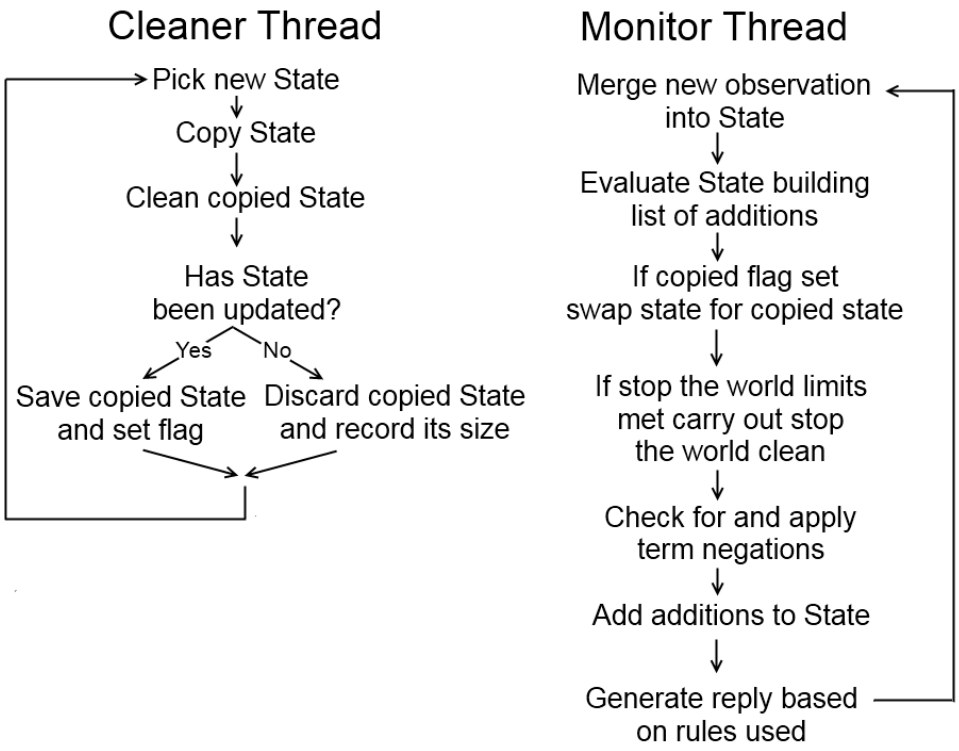


Figure 5.4: The cleaner thread and monitor threads do not block each other, instead the cleaner thread two atomic boolean values are used to ensure no terms are lost from the state.

Each state in the frontier, or single state, is registered with the cleaner. The cleaner attempts to clean each state in turn. This process is outlined in Figure 5.4 and involves the cleaner thread copying the state and removing all garbage and redundant terms from the copy. If the state has not been updated whilst the cleaner was cleaning the copy is saved. If the state has been updated during the cleaning process the number of removed terms is recorded but the copy is discarded.

Sometimes the time taken to clean a state becomes so large the state is always updated during cleaning. To deal with this there is a *Stop the world* option that detects when the state has become unbearably full of garbage or repeated terms based on the recorded number of removable terms. This blocks the monitor thread and carries out the appropriate cleaning. There is also an *intense garbage flag* which allows the user to turn on per step garbage collection which checks for garbage as the state is evaluated, this is useful if the user knows there will be a lot of garbage terms produced.

The cleaner thread has a number of parameters and choosing these can be very important for the smooth running of the monitor. The values given here were developed through experimentation. As mentioned later, in section 8.3, these parameters would ideally be dynamic, changing in response to the behaviour of the monitor.

Name	Description	Default value
sizeRemoveLimit	The size of state over which object removal is left to the cleaner thread.	100
stopTheWorldHardLimit	When the number of potential removed terms reaches this limit a stop the world clean will be invoked.	100
stopTheWorldPercentageLimit	When the number of potential removed terms as a percentage of the state size reaches this limit a stop the world clean will be invoked	50
restTime	The number of milliseconds the cleaner thread rests in between iterating over its store of states.	100

The rest time value is required to reduce the interference with the monitor thread - especially in the case of a small single state. However there will always be some interference of the cleaner thread with the monitor thread. The synchronisation overhead is very low but a hidden cost will come from sharing the state data structures in memory.

5.3.3 The Tidy Method

In version 2 of RuleR the evaluation of an observation is carried out by a `step` method. In this optimisation the book-keeping part of each evaluation from the `step` method is factored out into a separate `tidy` method. A call to `dispatch` calls `step` and then `tidy`, a call to `fastDispatch` just calls `step` and a separate call to `tidy` is required. The monitor wrapper calls `fastDispatch` first, returning the reply to the program before calling `tidy` and moving on to the next event.

The only book-keeping work currently in the `tidy` method is the logging of frontier sizes for performance monitoring. Ideally more work could be moved to this method to allow a quick turn around.

5.3.4 Parallel Versions

The final approach taken is to implement a number of parallelisations of the RuleR algorithm. These concentrate on the evaluation of the frontier. Recall that to evaluate an event each rule part of each rule activation of each state is evaluated for that event and these operations can be executed independently.

A data parallel fork/join approach is taken where the evaluation is split into a number of subtasks by segmenting the data structures to be evaluated. These subtasks are then executed in parallel and then the results collated. Two basic methods of segmentation and three methods of execution are implemented, and then for the last method of execution two further segmentation methods are developed - this gives eight parallel versions.

Segmentation - In the non-deterministic RuleR monitors the frontier is segmented, in the deterministic RuleR monitors the single state is segmented. The two basic segmentation methods are

In place segmentation just calculates the ranges and passes the whole data structure to the subtasks. This means the data structure is shared, which may lead to memory access issues, but involves a minimal segmentation cost.

Sublist segmentation creates individual sublists from the data structure for each subtask. This segmentation cost will scale to the number of subtasks and the size of the frontier.

In respect to load balancing - the act of attempting to ensure each thread carries out a similar amount of work, a special `getBalancedRules` method is included in the state data-structure which evenly mixes rules from different rule types. The reasoning being that spreading similar rules across subtasks will increase the chances of each subtask containing the same number of firing rules. Another future direction would be to explore more advanced methods for load balancing.

To address the issue of segmentation costs outweighing gains through parallel execution there is a limit on the size of the frontier or state regulating when that frontier or state is to be evaluated in parallel - for the deterministic rule monitors this is 10 times the number of threads and for the non-deterministic rule monitors this is 2 times the number of threads (these numbers were selected after experimentation). As explained in section 8.3 this would ideally be dynamic.

Execution - The three execution methods are

Basic Threads - This approach executes each subtask in a new thread - the frontier or single state is segmented and for each segment an object implementing Runnable is constructed and ran. This object carries out the apply method of the standard rule monitor and deposits the result in a shared list data structured. The monitor thread then calls `join()` on each created thread, waiting for each to complete. The threads synchronise on the list data structure which they add their results to when finished and this list is evaluated in serial by the monitor thread once all threads have completed.

Java Thread Pool - Instead of creating n threads on each step a thread pool is created at the beginning with n threads in it. This removes the cost of thread creation but introduces the cost of communicating with the thread pool object. A Callable object is created for each segment and passed to the thread pool. To avoid having to construct new objects on every step a version where a pool of objects was created and used was implemented but this saw no improvements so was discarded.

Fork/Join Library - The first two approaches are more suitable for large, long lived, tasks. As described in section 3.2.1 Doug Lea has developed a Java Fork/Join Framework [42] which is more suitable for small tasks generated in a fork/join manner. In addition to the in place and sublist segmentation methods above *fine grained* and *recursive* segmentation are also used for this approach.The fine grained implementation creates a task for each element in either the frontier or single state. The recursive implementation creates tasks recursively - for each state in the frontier, then each rule activation in each state and then each rule part in each rule (or for a single state each rule activation in each state and then each rule part in each rule). For each segment a task object must be created and results are collated by the monitor thread afterwards.

To implement these different versions a choice and a number of threads is included in the RuleR constructor, which are passed to the parser. If the number of threads is 1 the choice is ignored and a non-parallel version of the monitor is created, otherwise the relevant parallel implementation of the monitor is created - Dp, NDp, DSOp or NDSOp.

5.4 Testing the Framework

To ensure that the framework and underlying RuleR monitor is running correctly a suite of Junit tests is created. To do this a number of specifications are used which explore all of the RuleR functionality the developed version can handle. Due to time constraints some functionality has not been implemented - such as the use of sub rules.

To help get a good picture of where errors are occurring whilst setting up these tests a debug mode was added to the RuleR object. This records the events passed to the monitor and lists them if the monitor fails, giving a clearer picture of what happened than merely reporting the point of failure.

5.5 Summary

This chapter presented the implementation of an experimental framework for monitoring asynchronously and an optimised parallel RuleR monitor. The next two chapters explore the success of these implementations.

Chapter 6

Experimental Results

This chapter presents a number of microbenchmarks designed to explore interesting properties of the framework. These microbenchmarks have been designed to establish the scope and limitations of the developed framework and optimised monitor. The next section presents an evaluation of the framework as applied to the international DaCapo benchmarks. The specifications used in these microbenchmarks can be found in appendix B.1, where a key is provided.

6.1 Experimental Setup

For reproducibility a brief outline of the experimental setup used in this project is given.

The machine used in this project is an Apple Mac Pro, as described in section 3.4, running Mac OS X 10.6.4. The version of Java used was 1.6.0_20. The project was developed under version 3.5.2 of Eclipse using version 2.1.0.e35x-release-20100630-1500 of AspectJ. Unless otherwise specified the JVM options given were `-server -Xmx10240M`, giving the program 10GB of memory, setting the JIT compiler threshold to 10,000 and using the ParallelGC garbage collector.

6.1.1 Metrics and Graphs

It can be difficult to decide what metrics to use to evaluate results - merely reporting running times does not often allow for good analysis. Speedup is commonly used as a metric - one problem with this metric is knowing whether the base time used is the serial code or the parallel code running on one thread. Occasionally self-speedups will be given to demonstrate scalability. This report mainly uses the metric of temporal performance, the reciprocal of time. This means that on graphs higher points represent better performance. Temporal performance is chosen as it makes results more visually digestible whilst still allowing the reader to calculate the results from the graph, which speedup does not.

The majority of graphs given here and in the next chapter include error bars indicating the range of results, occasionally the standard deviation will be given in the text if this is of particular interest. All code is ran until times converge, displaying confidence that all JIT compilable code has been compiled, and all times presented are a mean average of at least three runs.

6.2 Looking at Different Parallel Versions

In section 5.3.4 eight different parallel implementations were developed to evaluate the single state in a deterministic RuleR monitor, or frontier in a non-deterministic RuleR monitor. This microbenchmark compares these different versions. A deterministic workload is used where the single state is expanded to a certain size and on each step half the rules are fired whilst the single state remains the same size.

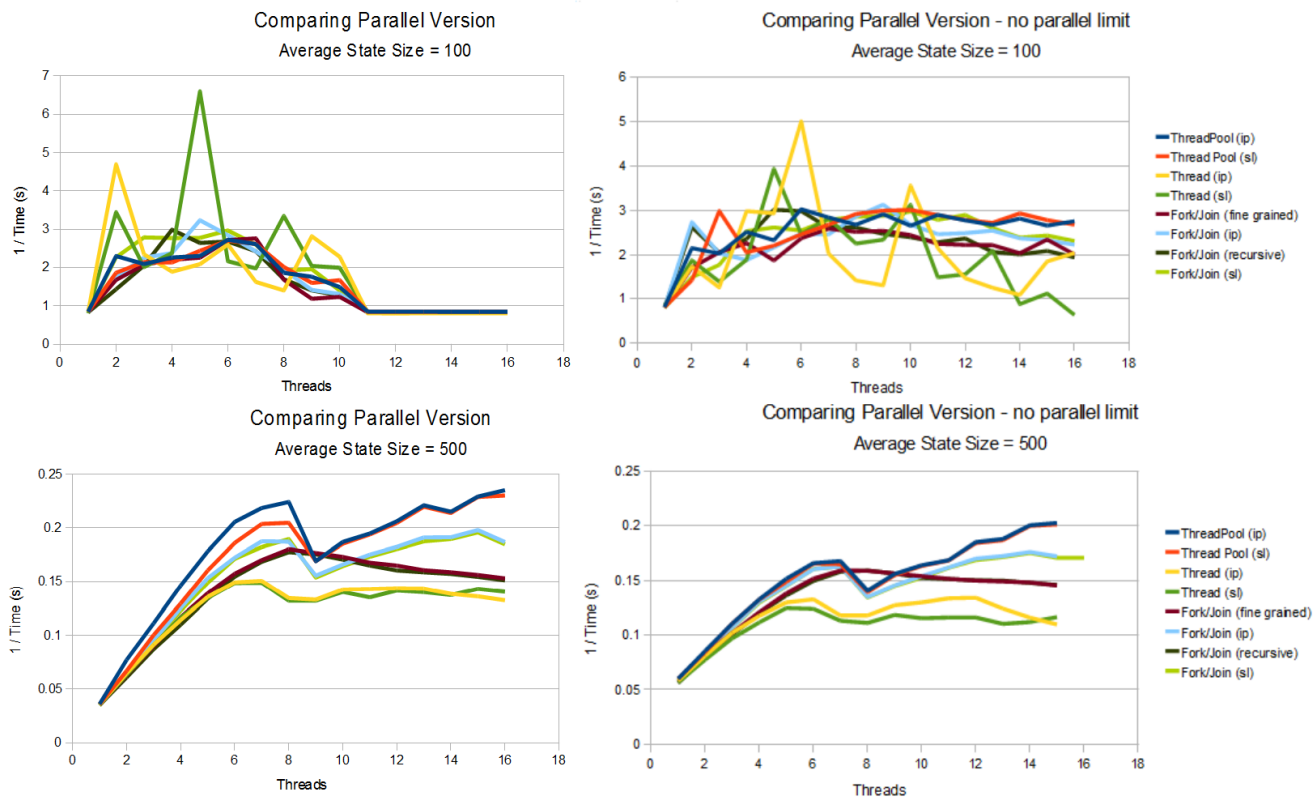


Figure 6.1: Comparing the eight different parallel implementations for evaluating the frontier. These results are for the DSO (Deterministic Single Observation) RuleR monitor for different average sizes of the single state and with and without the standard parallel limit. Error bars have been omitted due to the large number of results being compared. For the average size of 100 elements the standard deviation is typically around 10% but becomes as high as 60% in some cases, for 500 elements standard deviations of a few percent are typical - the worst culprits for large variance are the basic threads versions.

6.2.1 Results

Figure 6.1 shows the results for running the eight different versions for two different problem sizes with both the normal parallel limit (10 times the number of threads) and no parallel limit.

The first thing to note when comparing the two graphs for the smaller problem size is that it has an average state size of 100 elements and as the parallel limit is ten times the number of threads when this is set this workload will only be evaluated in serial when ran with 11 or more threads - this explains the sharp drop off at the end when the normal parallel limit is used. This parallel limit will also throttle other results - with 5 threads only half the work is carried out in parallel. It can be seen that when this

parallel limit is disabled performance increases across the board.

For the small workload the 'thread' approaches perform erratically and the results show a large standard deviation - between 40 and 60 percent. This will be due to dynamic thread creation and scheduling effects. Other results also fluctuate greatly as the small workload does little to spread out setup costs and other random effects are more pronounced. Some speedup is experienced with all versions giving similar results. The performance of all approaches seems to flatten off when moving above 8 threads as a lack of work sees scalability decrease - a maximum self-speedup of 3.69 is seen for the thread pool approach without a parallel limit, compared to a maximum self-speedup of 3.21 with the limit.

However the large problem size sees worse results when the parallel limit is removed - a maximum self-speedup of 6.39 is reduced to 3.61, this is due to the parallel limit forcing work best done in serial to run in serial.

For the large problem size there is an obvious drop in performance between 8 and 9 threads. This is because up until 8 threads each thread has been able to run on its own core, when using between 8 and 16 threads the hyperthreading technology is used - this allows two threads to run on the same core sharing the execution cycles.

The segmentation methods seem to have made little difference. For the large problem size up until eight threads the in-place method performs better than the sublist method for the Java thread pool versions, after this performance is the same.

For other methods the difference between the segmentation versions is not large, with the sublist costs probably overshadowed by other segmentation costs such as thread creation. The fine grained and recursive segmentation methods do not scale well beyond eight threads.

6.2.2 Summary

For the rest of this section the parallel version employing Java thread pools and the in-place segmentation method will be used as this scaled best in this and other experiments. The thread creation approach was not as expensive as expected and outperformed the Fork/Join library, which is probably more suited to smaller recursive, tasks. The fine-grained and recursive segmentation approaches were not given much opportunity to use their advantage in this workload, future explorations into workloads using large complex, rules would be useful to establish any potential benefits these segmentation methods may have.

6.3 Deterministic Large Problem Size

This, and the next, microbenchmark measure the effects of the parallelised RuleR engine for large problem sizes. This microbenchmark does not use the framework and uses manual instrumentation. The aim of this microbenchmark is to measure **scalability**, this is very important as larger machines with more and more cores are being developed and unless an application can scale to these larger number of cores its usefulness is limited.

This microbenchmark is concerned with a deterministic large problem size, and therefore the deterministic single observation version of the RuleR monitor used. This microbenchmark first looks at the

effects of scaling the number of threads used, and then looks at the effects of using different work sizes.

Four different workloads are used, outlined in table 6.1, where the number of rule activations fired on each step and number of rule dependencies vary. This makes a difference as firstly workloads which fire more rules and depend on more rules carry out more work per rule activation in the frontier, and secondly rule dependence can lead to memory contention.

	setup			iterate		
	average size	fired	dependent	size	fired	dependent
1	$n/2$	2	0	$n+2$	1	0
2	$n/2$	2	0	$n+2$	$n/2+1$	0
3	$n/2$	2	0	$n+2$	$n/2+1$	n
4	$n/2$	2	0	$n+2^*$	$n/2+1$	$3n$

*($n/2$ with 4 parts)

Table 6.1: Workloads used for the deterministic large problem size microbenchmark. Given the parameter n this table divides the workloads into setup and iterate parts, detailing the size of the state, number of rules fired and number of rules depended upon for each step in that part for each workload.

6.3.1 Scaling Threads

The results are presented in Figure 6.2 for each of the workloads with the maximum single state size set at 500 with 1000 iteration steps, making the overall number of steps 1250. The last three workloads achieve some improvement but the more intensive workloads see greater gains whereas workload one sees an overall decrease in performance when ran in parallel.

Workloads one and two see a noticeable decrease in performance on the movement to two threads indicating that the overheads outweigh the benefits of parallel execution at this point.

To explain the differences in performance the differences between the workloads should be examined, and it can quickly be concluded the only difference is the amount of work being carried out on each step. To begin with the only way to explain the increased performance between workload one and two is that workload two fires $n/2$ more rules on every step than workload one. The large difference in performance between workloads two and three must be due to the fact that workload three has $n/2$ rules firing in the iterate phase that each depend on two rules. And workload four achieves much greater scalability than the other workloads as it is carrying out much more work per step - the state must be searched $3n$ times on each step and half of the rules have 4 parts (meaning that they will take four times as long to check as the other rules).

From this we can conclude that the amount of work being carried out in each step significantly affects the effectiveness of parallelising this algorithm. This makes sense as there will be a certain overhead on each step involving segmenting the work, communicating with the thread pool, collecting results, as well as hidden overheads related to memory access - for the work to see an increase rather than a decrease in performance the amount of work being carried out in parallel must be large enough so that the gains from executing it in parallel outweigh the overheads involved.

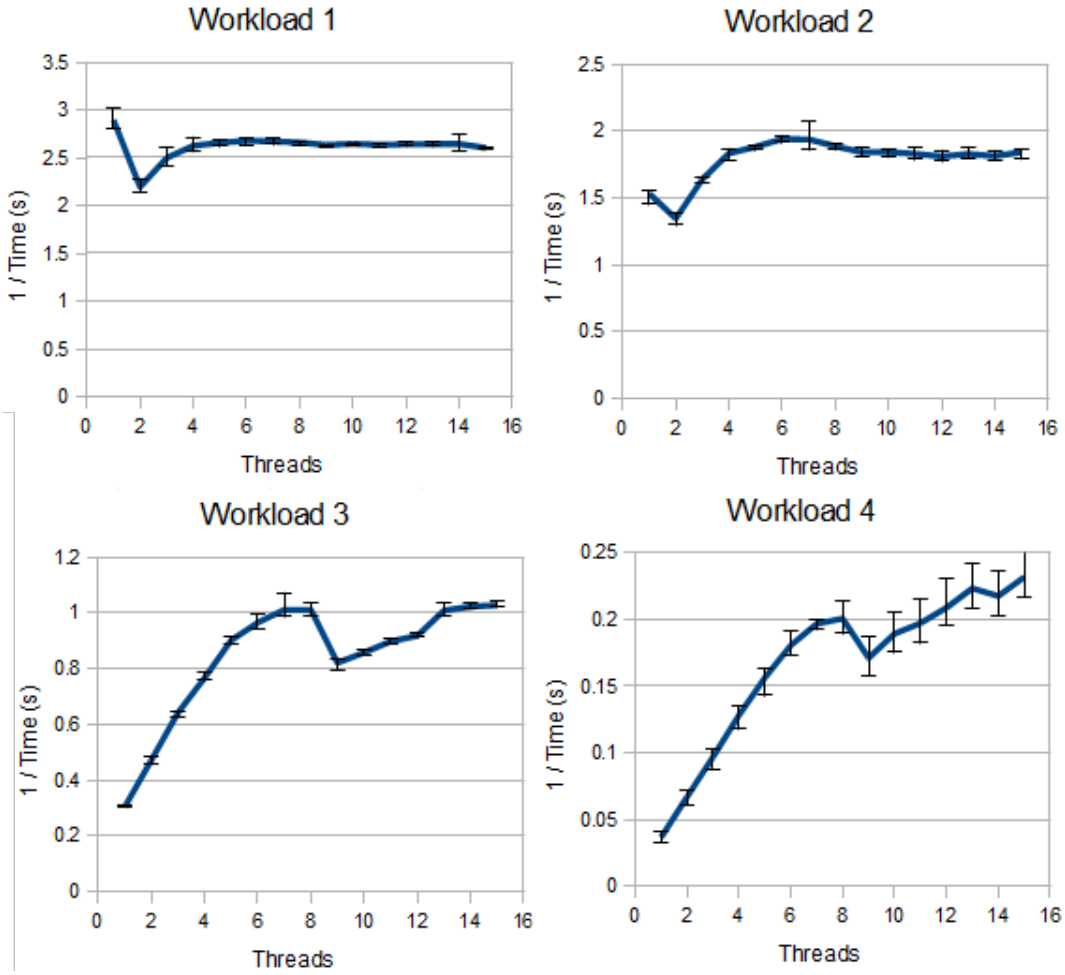


Figure 6.2: Results for four different workloads with a deterministic large problem size, looking at scalability. Temporal performance used (larger is better).

Workload four shows the greatest scalability, as previously mentioned this is due to a larger portion of the overall work being carried out in parallel. The best result for workload 4 is a self speedup of 6.62 when running on 8 threads was seen. Workload 4 also saw the greatest fluctuation in results this will probably be a combination of memory access and load imbalance overheads. As workload 4 has many more rule dependencies each thread will need to access the same data structures, increasing contention, but this contention will not be consistent as it is very dependent on the ordering of terms in the state and the division of these terms among threads. This leads on to the topic of load imbalance, of the 500 terms in the state some of them will take a lot longer to evaluate than others - not because of the way they are spread out (as for this workload the load balancing getRules method spreads these out evenly) but because when searching for dependent rules some rules will be found much sooner, again this imbalance is not consistent.

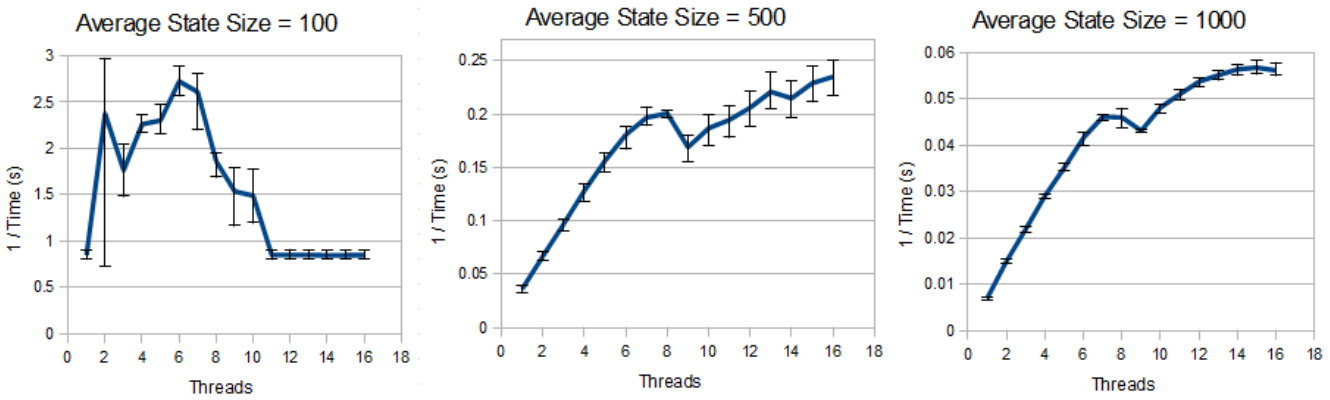


Figure 6.3: Results for workload four for different work sizes.

6.3.2 Scaling Work Size

Using the most work intensive of the workloads from above, workload 4, the effects of scaling the work size are examined. Figure 6.3 presents the results for n=100, 500 and 1000. For n=100 results for 11 threads and above may be discounted due to the parallel limit forcing the workload to be evaluated in serial.

The scalability increases with the work size - a maximum self-speedup of 3.21 was seen for n=100 (using 5 threads) but this increases to 8.13 for n=1000 (using 16 threads). The second two graphs see a dip at 8 threads before performance is seen to climb back up again and increase beyond that seen at 8 threads. This dip could be attributed to a movement to hyperthreading as the experimental machine has 8 physical cores (grouped as 2 processors) each capable of running 2 hardware threads.

However another possibility is that it is a memory effect - the first eight threads may have been scheduled on the same processor and moving above 8 threads caused the second processor to be used, meaning that the data structures needed to be communicated between the two processors (this takes longer than communication between cores). As more threads run on the second processor this overhead becomes less significant. Time constraints did not allow for this second idea to be verified.

6.3.3 Summary

This microbenchmark shows that evaluating the single state used to hold rule activations in the deterministic case scales reasonably well for large problem sizes if the amount of work done per step is large enough. This scalability is seen to increase with the work size and two possibilities were suggested for why performance drops slightly beyond 8 threads.

6.4 Non-Deterministic Large Problem Size

This microbenchmark uses a non-deterministic large problem size to examine the scalability of the optimised RuleR monitor. This is considered separately from the deterministic case as the optimisations applied here and the way the frontier is divided is quite different than in the deterministic case.

Again four workloads are used, to create these the workloads of the previous section are taken and the state is duplicated a number of times.

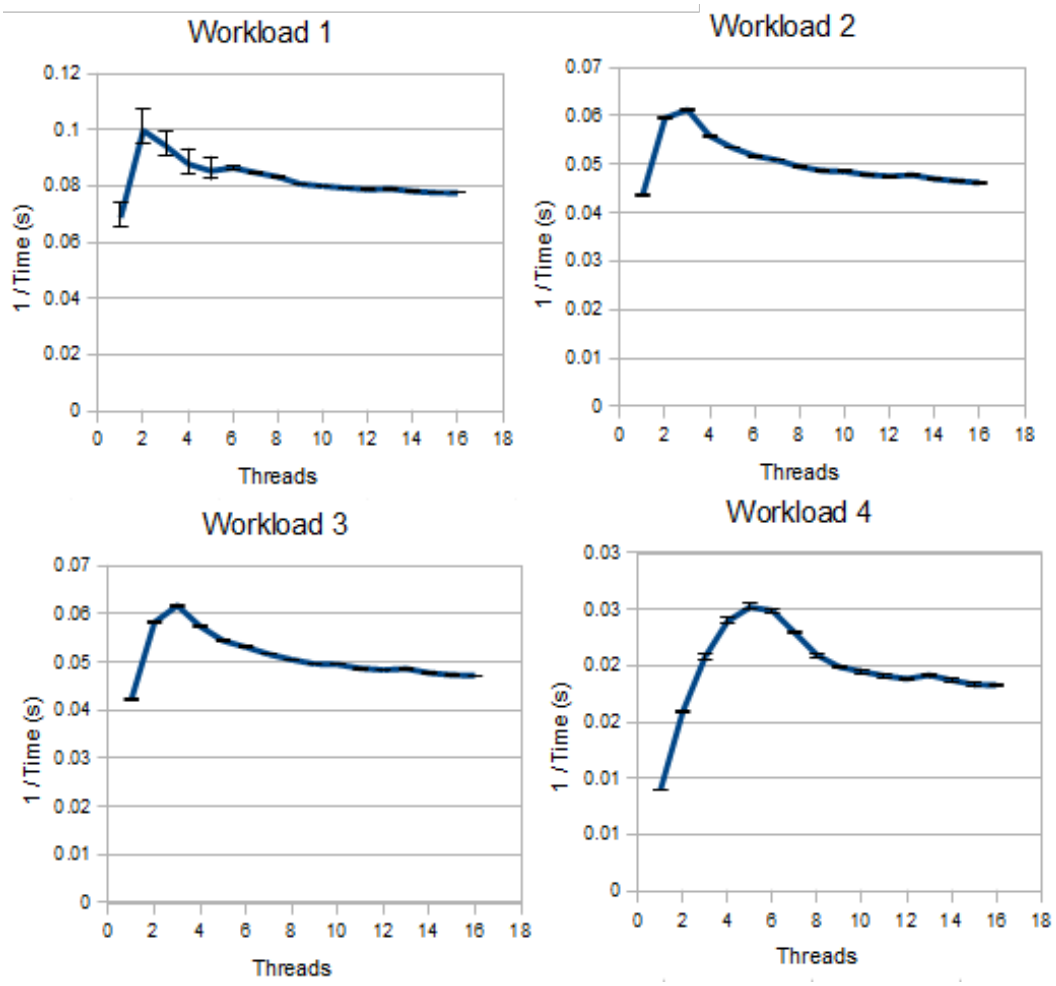


Figure 6.4: Results for four different workloads with a non-deterministic large problem size, looking at scalability. Temporal performance used (larger is better).

6.4.1 Results

The size of each single state is fixed at 10, the number of states fixed at 256 and the number of iterations fixed at 1000. This means that there are 1013 steps overall.

The results of running each of the workloads is given in Figure 6.4. Firstly the fluctuations in results seemed surprisingly small but as each state represents the same amount of work and exists independently in memory the causes of fluctuations discussed in the previous section do not exist here.

Similarly small improvements are seen in workloads 1-3 with performance increasing for 2-4 threads before tailing off. Self speedups of around 1.5 were seen for these workloads but the overheads involved in parallelising this workload must be larger and more dependent on the number of threads than for the workloads seen in the previous section.

Workload 4 presents better scalability - reaching a self-speedup of 2.78 for 6 threads, again this tails off with more threads. Again these differences can be attributed to the amount of work being done per step.

6.4.2 Summary

The parallel non-deterministic version of the RuleR monitor did not scale as well as the deterministic version for the inspected workloads. A maximum self-speedup of 2.78 was observed here using 6 threads. However fluctuations in results were very low due to the frontier being decomposed into collections of states where each state represents the same amount of work and does not introduce load imbalance or memory access overheads.

6.5 Clustered Workload

In section 4.2.2 one of the particular workload organisations identified was the clustered workload, where a cluster consists of a number of temporally close events with the first event being a significant one, and the workload consists of clusters spread out throughout the monitored application. This workload presents itself in the `hasNext` property - the property that for a given iterator object every `next` call on that iterator is preceded by a `hasNext` call. This workload will also be common in similar datastructure-focused specifications where the data structures are created reasonably infrequently but used heavily once created.

The number of iterators is fixed to ten and the number of iterations on each iterator is varied, to alter the size of a cluster. To simulate the time between clusters the main thread is made to wait using `Thread.sleep(n)`, where n is the number of milliseconds to wait. There was an attempt to simulate wait times using random work calculated to the same time but whilst similar results were achieved on average the results varied greatly.

6.5.1 A Model

A model relating wait times and number of events is created. This is then used to predict the minimum wait time required for a given number of events to allow the framework to hide part of the monitoring overhead, or the maximum amount of work a particular wait time can cover. First a model to describe the behaviour of the base RuleR monitor is constructed, then a model to describe the behaviour of the clustered approach using the framework.

In the specification being used there are three different kinds of dispatch messages - `hasNext`, `next` and `end`. The `end` message is sent as the last message of a cluster and, assuming correct behaviour, the other two messages are sent alternately starting with `hasNext`. For n iterators $2n+1$ events are generated - n `hasNext` messages, n `next` messages and 1 `end` message.

RuleR Monitor - When using the RuleR monitor by itself all messages are sent directly to the monitor and evaluated immediately. Each of the events may take different times to evaluate but instrumentation

costs will be the same. Making the model

$$((2n+1)*I) + n*E_{hasNext} + n*E_{next} + E_{end} + W$$

where I stands for instrumentation costs, $E_{message}$ stands for the time to evaluate a message of that kind and W stands for wait time.

Framework: Clustered Approach - When using the framework all messages from the cluster, apart from the first message, are sent in a non-blocking manner. The first `hasNext` message of a cluster is sent in a blocking manner. If the previous cluster has not finished being evaluated when the first message of the next cluster is dispatched in a blocking manner then the application must wait for this to complete. Therefore the time taken per cluster is the time to execute the blocking call plus either the time to dispatch all of the messages plus the wait time or the time to evaluate all of the messages, whichever is greater. This can be expressed as

$$MAX((((2n+1)*(I+D_{nonBlocking})) + W); (((2n+1)*I) + n*E_{hasNext} + n*E_{next} + E_{end})) + D_{blocking}$$

where D_{mode} stands for the time to dispatch a method using that dispatch mode.

Rough measurements are taken for the variables giving an inequality involving n and W. Note that the values $E_{hasNext}, E_{next}$ and E_{end} will vary for the two different models as the monitor in the clustered approach runs in a separate thread leading to a memory access overhead between threads. By solving the inequality the point at which the clustered approach is no longer beneficial can be predicted, this calculated point is included on the graphs below.

6.5.2 Results

Figure 6.5 shows the results for a number of different lengths of wait time. As expected for each wait time as the number of events increases the clustered approach becomes less effective and eventually takes longer to run than the standard monitor approach.

The crossover points predicted by the model are shown on the graphs. In the case of 5 and 10 millisecond wait times these predictions were reasonably accurate. For 1 and 50 millisecond wait times the predictions underestimated and overestimated the performance of the clustered approach respectively. Two likely causes for this are firstly in the 1 millisecond case the model may not account for certain setup costs which become significant with fewer events, and secondly in the 50 millisecond case when more events are being used the memory effects may have an effect on the average time to process as an event with tens of thousands of events being processed even a small change will cause a change in the model.

6.5.3 Summary

The clustered approach achieves minimal positive results giving around 1.2 - 1.5 times self-speedup depending on worksize and wait times. This approach does not scale as it relies on the monitor being run

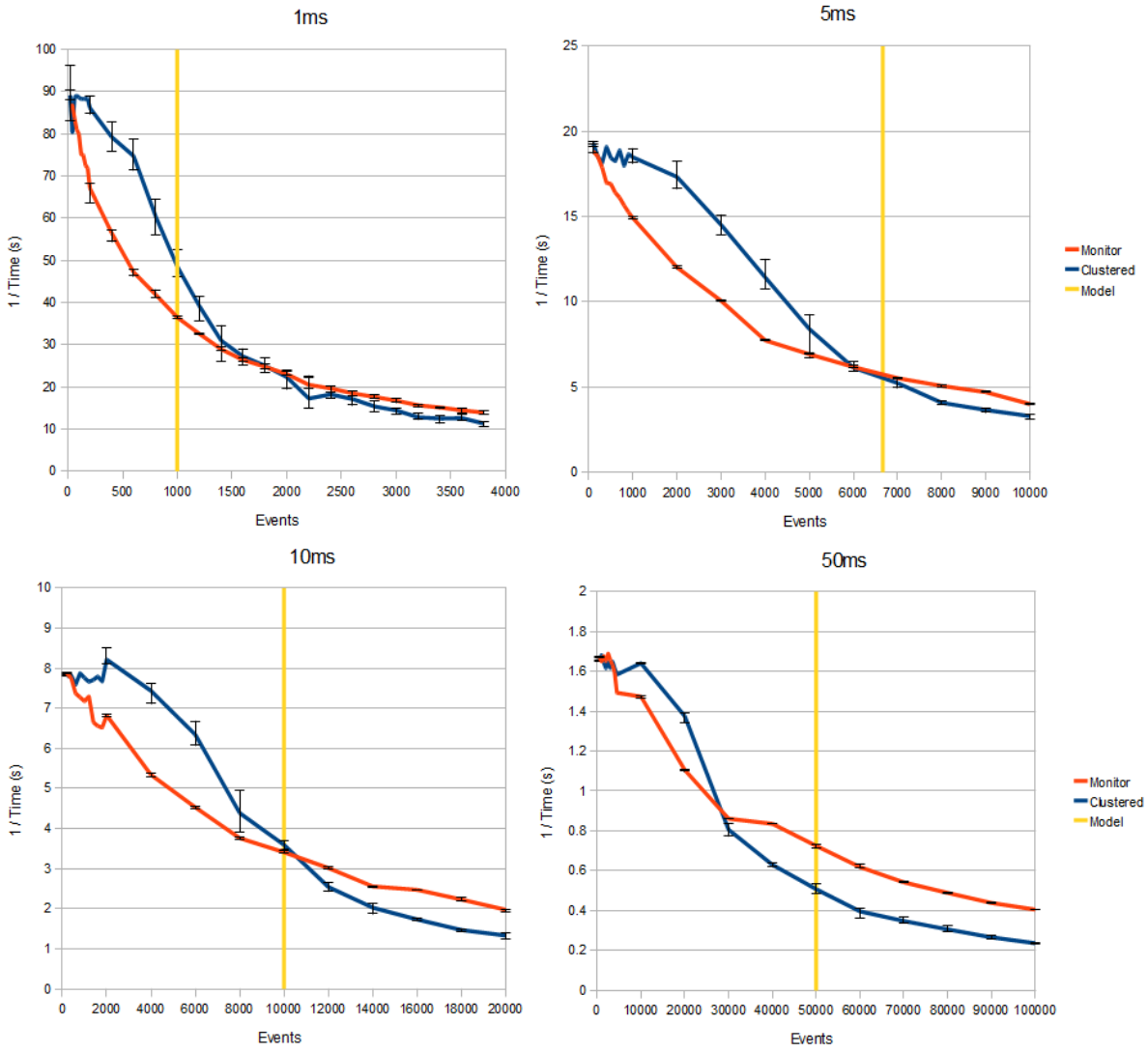


Figure 6.5: Results for the clustered workload microbenchmark.

in a single separate thread. It should be noted that the amount of work being carried out by the monitor for each step is minimal in this microbenchmark. It would be interesting to also examine the benefits of this approach with more work intensive properties.

6.6 Taggable Workload

Another workload organisation identified was the taggable workload, where the execution trace can be divided into a number of independent sequences of events with each sequence identifiable by some unique tag. Two different workloads are examined - the `hasNext` example and a workload with heavily interleaved events. The effects of this approach should scale with the number of threads given.

6.6.1 `hasNext` Workload

To begin with the taggable approach is compared to other approaches whilst scaling the number of iterators used, then the effects of scaling the number of threads is examined.

Comparing to other approaches - When there are many iterators the taggable approach should be more efficient than the standard or clustered approaches. Figure 6.6 compares the tagged approach, running with ten threads, with the other approaches. The graph shows that for this workload the tagged approach always runs faster than the clustered or basic monitor approach, as well as the original RuleR version 2 monitor. In this case the clustered approach performs badly as the the time between clusters is too small to make up for the overhead.

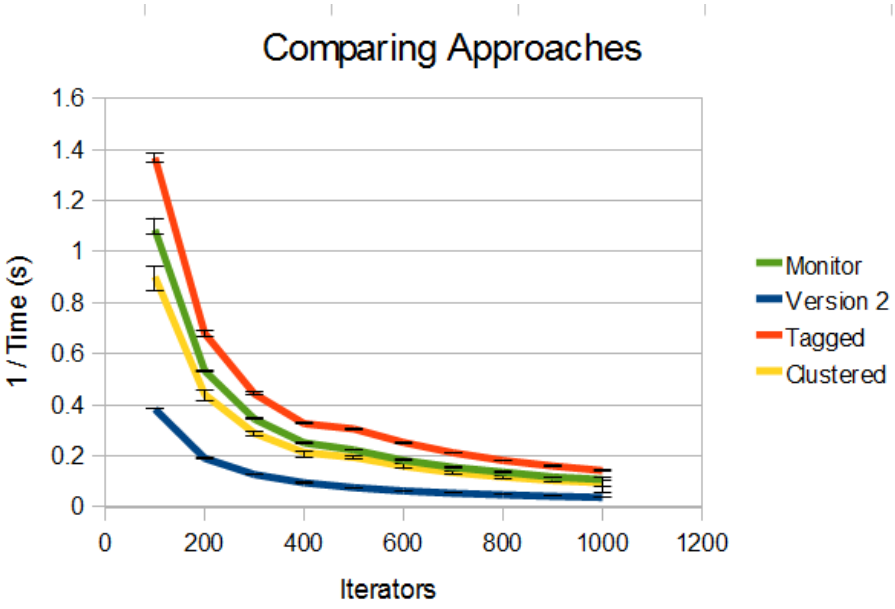


Figure 6.6: Scaling the number of iterators when comparing taggable, clustered and standard approach. Running time is reported (lower is better).

Scaling Threads - Using more threads should give better performance, but the gains achievable will be limited by the number of iterators as if the execution trace can only be decomposed into four parts then using eight threads will not give any better performance than using four. Figure 6.7 shows how the tagged approach scaled for different numbers of threads and different workloads.

Generally this approach did not scale well - the maximum self speedup was 1.48, which was with 100 iterators each performing 10,000 iterations monitored using six threads. In most cases a small benefit was gained by using two to four threads.

In the two top graphs for 10 and 100 iterations the performance is erratic with run times varying greatly. This is because as each iterator does not do much work it is important that the iterators are evenly spread between monitors. The naive hash function used takes the hash code of the iterator modulo the number of monitors.

In the case of 5 and 10 iterators little scalability is seen - again due to the naive hash function. Firstly above 5 and 10 threads respectively there were monitors running without any work to do. Secondly if the hash function did not evenly spread the work then one or two monitors could have been doing all of the work. A small experiment was carried out to ensure that the hash function gave reasonable spread - which it did do for a reasonable number of tags.

Looking at the top four graphs, where 100 iterators were used, it can be seen that before 10,000 iterations minimal improvement is seen. For 10 and 100 iterations some improvement is shown if considering just the mean values but it can be seen that this varies greatly - the standard deviation for both cases is also large. It is important to realise that the tags were being used in order so a trade-off exists between the number of iterators and number of iterations - both contribute to load balancing.

Comparing the 10 iterators with 10,000 iterations case and the 100 iterators with 10,000 case it looks like the number of iterators (tags) places a limit on the scalability and the amount of work per iterator (tag) controls the speedup obtainable, this is supported in the next section.

6.6.2 Interleaved Events

In the `hasNext` example the tags decompose the execution trace into a number of smaller traces occurring in order and without interleaving, here a workload with interleaved events is presented. This workload uses m objects and n tags, with each tag relating to an object. The specification counts the number of events involving each object and the instrumentation tags each dispatch with the hashcode of the object it relates to.

The results are given in Figure 6.8 for different values of m and n. It can be seen that for very large workloads involving over a million objects near linear speedup is achieved. There is a large variability in the results - this is again partly due to the naive hash function creating load imbalance.

As is suggested in the previous section it can be seen that when there are more tags the limit on scalability goes up and when each tag has more work the magnitude of this speedup increases.

6.6.3 Summary

The tagged approach applied to the `hasNext` workload saw minimal scalability and for most workloads saw performance decrease given more threads. However for the interleaved workload near linear speedup was observed for large well divided workloads. This approach seems promising, offering good scalability.

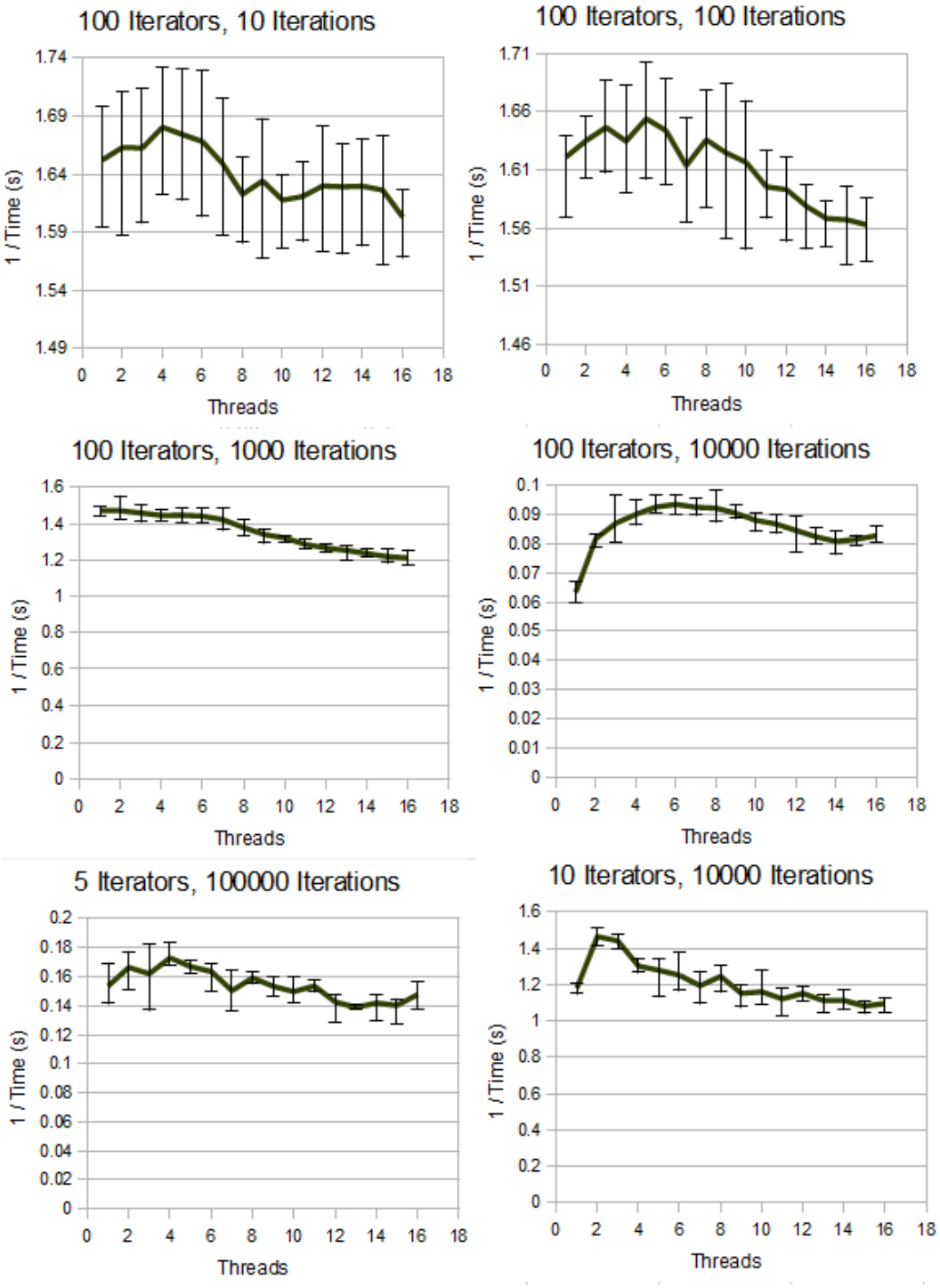


Figure 6.7: Scaling the number of threads using the tagged approach for different numbers of iterators and iterations per iterator for the `hasNext` property.

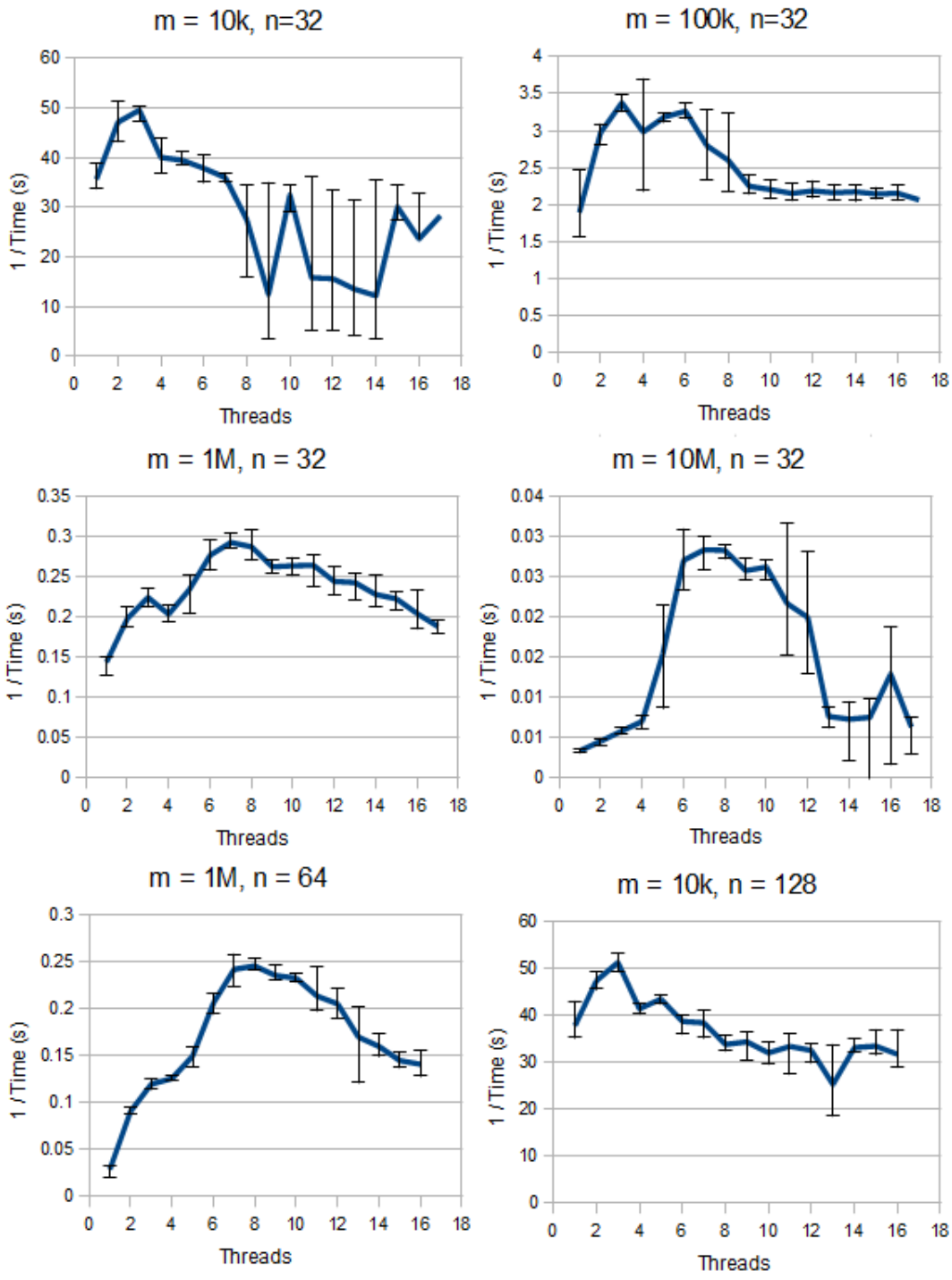


Figure 6.8: Scaling the number of threads for the interleaved workload using the tagged approach.

6.7 A Multithreaded Workload

This microbenchmark examines the different approaches taken to dealing with per-thread properties. A per-thread property is one where events from different threads can be evaluated independently. Within the framework there are two different approaches to monitoring per-thread properties and without the framework the monitor can deal with per-thread properties directly.

The clustered workload from above is used here - a number of threads are created each running the same amount of work. The first approach within the framework is to run separate monitor wrappers for each thread, this is labeled as 'Framework' as it occurs within the framework without effecting the monitor wrappers. The second approach within the framework is to hold separate queues within the monitor wrapper for each monitored thread - there then exist a number of strategies for evaluating these queues. The monitor by itself can create independent data structures for each monitored thread, allowing multiple threads to operate over the monitor concurrently.

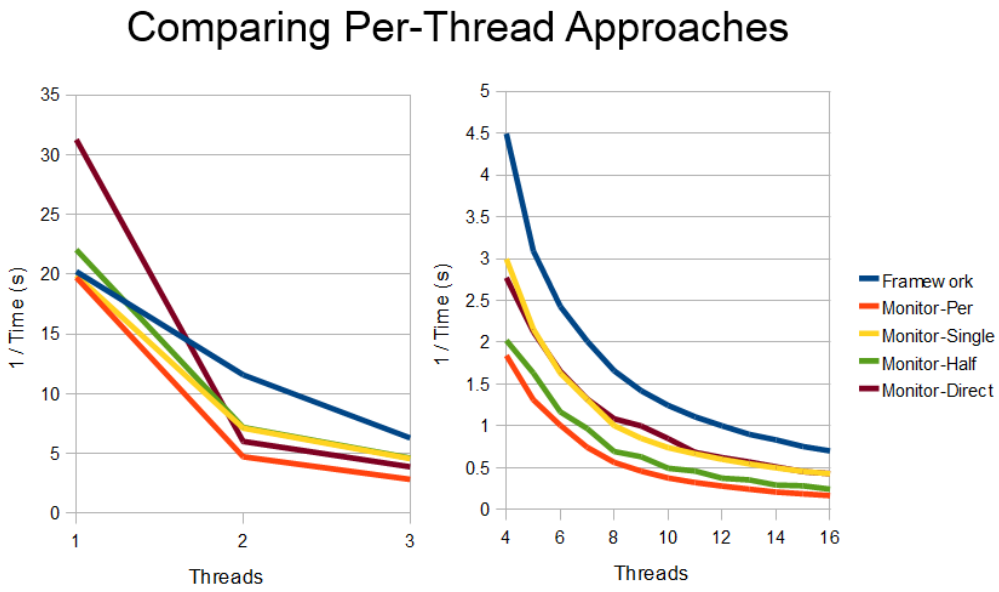


Figure 6.9: Comparing the in monitor and in framework per-thread strategies.

Figure 6.9 shows five different approaches - the framework approach described above, three different strategies for the monitor wrapper approach and the approach using the monitor directly. The three strategies used with the monitor wrapper are Per - a thread is created in the monitor wrapper for every monitored thread, Single - a single thread is used to monitor all events, and Half - half the number of monitored threads are used to evaluate the events.

For a single thread the direct approach performs best as it incurs no additional overhead. The monitor wrapper Per approach consistently performs worst - under-performing the single thread and direct approaches, the monitor wrapper Half approach does not fare much better. The monitor wrapper approaches incur a significant synchronisation cost as a shared data structure holds the different per-thread queues. For two or more threads the framework approach consistently performs best, although this may potentially cause interference problems as many threads are created.

In summary the in framework approach is generally the best way to monitor multithreaded programs but if there is a worry that creating many threads will interfere then a single thread should be used within the monitor wrapper.

6.8 Changing the Specification

This microbenchmark inspects the effects of using different specifications to evaluate the same property. For this the familiar `hasNext` property is used. Four different specifications are used : the original specification that just records `next` and `hasNext` events; the end specification used throughout this chapter that uses an `end` event, generated when a hasNext on an iterator fails, to remove an iterator from the frontier; a lazy specification that only keeps track of one iterator at any one time; and a set specification that uses a set to extend the lazy specification by keeping track of previously seen iterators.

For comparison purposes the workload and instrumentation is fixed for each specification. Figure 6.10 gives the results with times normalised to those of the original specification. Two workloads are used - a normal one using many iterators in series and a nested one which makes heavy use of nested iterators.

The lazy specification fails on the nested workload. Even though the lazy specification should perform better, as it does less work in some areas, it must actively remove the last iterator when a new one is found which makes up for this. The set specification incurs a great cost as the use of sets in RuleR is expensive - even though the set is never used in the normal workload the larger data structures slows down the whole monitoring process.

This is just a short example to show that different specifications used to monitor the same property can perform very differently and highlight the importance of designing the specification carefully.

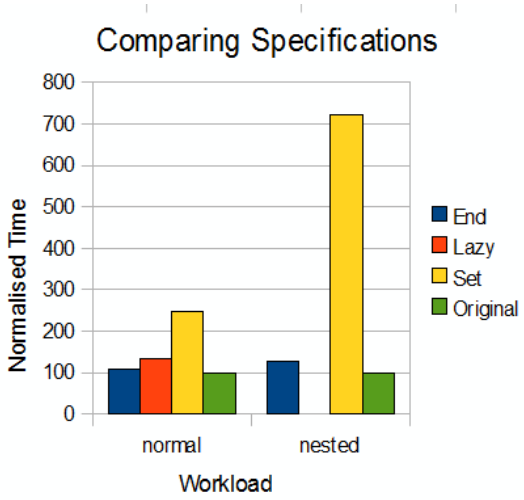


Figure 6.10: Comparing performance for different specifications.

6.9 Timing Limits

This microbenchmark examines the timing limits available under different monitoring approaches.

To measure the smallest measurable timing property a specification as given in appendix 6.9. Two events are sent to the monitor - this firs is made up of two observations, `check(long)` and `time(long)`, and the second is made up of a single observation, `time(long)`. The `time` observations are added in by the monitor as timing mode is switched on. On receiving the first event a rule is set up waiting for the second, the property will then fail if the first time plus the check time is less than the second time.

This means that decreasing check times can be fed to the monitor, recording the observation on which it fails. This is done for a number of monitoring approaches and single state/frontier sizes and the results recorded in Table 6.2, times are given in microseconds.

Approach	Single state/frontier size			
	1	10	100	1000
RuleR-V2	(40,201,545)	(1005,1108,1230)	(33400,33890,34600)	-
Base Monitor	(5,5,5)	(10,10,10)	(5120,8640,9415)	(8170,8885,9415)
Blocking Framework	(5,5,5)	(10,48,760)	(6255,8896,9435)	(6215,8923,9650)

Table 6.2: The smallest measurable time for different monitoring approaches, results are given as *(minimum, mean, maximum)* in microseconds.

The first thing to note is that the timings from version 2 of the RuleR monitor become very high quickly and results for a frontier with a single state of size 10,000 were not obtainable. Using the base monitor times were kept low for single state sizes of 10 and 100 elements, but these shoot up when the number of elements goes up to 1000 elements.

It is interesting to note that when the single state was expanded to 1000 terms the smallest time recorded for the base monitor was higher than that recorded for the blocking framework. This is likely to be due to fortunate operation orderings and thread scheduling rather than an artifact of the code itself as only one run managed this performance. This demonstrates how variable running times can be - this can be problematic if accurate timing properties were to be monitored (but only for large states).

6.10 Summary

This chapter has looked at a number of small microbenchmarks with the aim of examining the scope and limitations of the developed framework and optimised RuleR monitor.

Some good results were obtained, and some less favourable ones. It was found that if enough work is carried out on each step then a deterministic workload exhibiting a large single state observes near linear speedup for up to 8 threads (the number of physical cores on the experimental machine). This was found to scale well with the workload size also. The non-deterministic version of the optimised monitor was not found to scale as well as the deterministic version.

The clustered approach obtained some performance advantages but this was limited by the work size and speedups were not observed beyond 1.5 times. The clustered approach is not scalable as it only relies

on the monitor being run in a separate thread.

The tagged approach was found to perform better than other approaches for a particular workload and it was concluded that the number of tags limits scalability and the amount of work done per tag dictates speedup.

It was found that the Java Thread Pool version was the best parallel version and that the 'in framework' per-thread approach was better than any of the approaches operating within the monitor wrapper.

Finally the newly developed approaches allow smaller real-time properties to be monitored than version 2 of RuleR indicating that interference has been decreased.

Chapter 7

Evaluation

This chapter presents an evaluation of the framework using the international DaCapo benchmarks. Whilst the previous chapter focused on an exploration of the framework's limitations and scopes this chapter concentrates on the applicability of the framework to real world problems.

7.1 DaCapo Benchmarks

In this section the international DaCapo benchmarks are used to evaluate how applicable the framework and parallel RuleR monitor developed in this project are to real world problems. The benchmarks and specifications are presented, followed by a presentation and analysis of the results.

7.1.1 Benchmark Details

The DaCapo benchmarks, as presented in [2], are a set of open-source client-side Java benchmarks. The benchmarks currently included in version 9.12 of the DaCapo benchmark suite are described in Table 7.1. The table gives all the benchmarks included in the suite but due to time constraints and technical difficulties only a subset of these were used.

7.1.2 Specifications

Due to the complexity of these benchmarks it is not possible to develop specifications based on any properties associated with the work they carry out. Instead generic specifications associated with data structures and concurrency are used.

The specifications, as outlined in Table 7.2 and given in appendix B.2, include eight specifications previously used to test the `tracematches` tool, found in [20] (and on the accompanying website), and one for lock-ordering developed from work in [58]. The eight specifications taken from the `tracematches` study were converted from the tracematches specification language and appropriate instrumentation was written. The specifications were based on the safe use of Java library data structures as outlined in the Java specification. The lock-ordering specification is based on a state machine representation of an algorithm developed by Stolz to detect unsafe lock ordering.

Name	Description	Threads	Size	Time (ms)
avrora	A set of simulation and analysis tools in a framework for AVR micro-controllers.	23	large	6797
batik	An SVG toolkit produced by the Apache foundation. The benchmark renders a number of svg files.	2	large	4695
eclipse	executes some of the (non-gui) jdt performance tests for the Eclipse IDE.	-		
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.	1	default	1346
h2	is an in-memory database benchmark, using the h2 database produced by h2database.com, and executing an implementation of the TPC-C workload produced by the Apache foundation for its derby project.	-		
jython	Interprets the pybench Python benchmark.	-		
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.	16	default	1845
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.	8	large	2003
pmd	Analyzes a set of Java classes for a range of source code problems.	16	large	5092
sunflow	A raytracing rendering system for photo-realistic images.	16	large	5077
tomcat	Uses the Apache Tomcat servelet container to run some sample web applications.	16	huge	
tradebeans	Runs the Apache daytrader workload "directly" (via EJB) within a Geronimo application server.	-		
tradesoap	Identical to the tradebeans workload, except that client/server communications is via soap protocols.	-		
xalan	Transforms XML documents in HTML.	1	large	7457

Table 7.1: The DaCapo benchmarks, descriptions taken from [2]. The size column relates to the fact that some benchmarks are provided with different sizes of workload - the size is used is given here. Generally multithreaded workloads are scaled to the available threads.

Name	Description
SyncAll	When performing an action on a synchronized collection involving another synchronized collection the thread performing the action should hold the lock for both collections.
SyncIteration	A synchronized collection should only be iterated over from within a synchronized block.
FailSafeEnum	An enumeration created from a collection should not have `next` called on it after the source collection has been updated.
FailSafeIter	An iterator created from an collection should not have `next` called on it after the source collection has been updated.
HashMap	The hashcode of an object added to a hash map should not be altered whilst the object remains in the hash map.
HasNext	Every `next` call to an iterator should be preceded by a `hasNext` call.
HasNextElem	Every `next` call to an enumeration should be preceded by a `hasNext` call.
LeakingSync	After a synchronized version of a collection is created the unsynchronized version should not be used.
LockOrdering	Detects unsafe lock orderings leading to potential deadlock.

Table 7.2: Specifications used to monitor the DaCapo benchmarks.

7.1.3 Performance Results

This section groups the specifications based on the kind of workload they present. Each group is looked at in turn - the appropriate approaches are selected and the results given and discussed.

Timings in this section are taken from an average of single runs of the monitored benchmark using the overall time to complete the run, not the time reported by the DaCapo suite harness. This is because for some approaches, such as tagged, the harness will report a running time before monitoring has completed (this difference reflects the interference of the tool and is examined in the next section).

HasNext and HasNextElem These two similar per-thread properties produce an execution trace which either presents the clustered or tagged workload. Therefore clustered, tagged and nonblocking approaches are compared, for the tagged approach scalability is considered. The parallel monitor is not used as the number of active rule activations will not exceed two unless nested iterators are used.

Table 7.3 presents the results for monitoring the `hasNext` property for the selected DaCapo benchmarks using a number of different approaches. None of the examined benchmarks made use of enumerations, therefore the monitoring of this property added no overhead.

The alternative approaches did not provide any significant improvements over the basic monitor. There were two instances where an improvement was seen, as indicated in bold in the table. But this improvement was minimal and comparable to the range of results seen. There is an obvious correlation between the total number of events and the magnitude of slow down observed.

Table 7.4 gives the number of iterators and average number of iterations per iterator and from this it can be seen why these approaches did not provide any improvements. Both the clustered and tagged approaches depend on a significant amount of work being per iterator.

HasNext										
Name	Bare	Serial		Clustered		Tagged			Nonblocking	
		T	S	T	S	T	S	Th	T	S
avrora	6797	177645	26.14							
batik	4695	7987	1.7	10278	2.19	**7118**	**1.52**	4	25858	5.51
fop	1346	41334	30.7	66347	49.2	78753	58.5	16	230796	171.46
luindex	1845	2271	1.23	3197	1.73	3782	2.05	4	2565	1.39
lusearch	2003	2623	1.3	3814	1.9	4112	2.05	1	6998	3.49
pmd	5092	75424	14.81	250200	49.14	164710	44.81	15		
sunflow	5077	38844	7.65	141080	27.79	236185	46.52	16	710636	139.97
tomcat	11029	26635	2.41	31032	2.81					
xalan	7457	10964	1.47	**10840**	**1.45**	11480	1.54	1	11991	1.6

Table 7.3: Results for different approaches applied to the `hasNext` property. T stands for Time in milliseconds and S stands for Slowdown from bare (unmonitored) runtime.

	Events				Average iterations
Benchmark	hasNext	next	end	**total**	per iterator
avrora	8705660	4838147	3878776	13543807	3.49
batik	47618	16636	30982	64254	2.07
fop	764554	458418	72628	1222972	16.84
luindex	218	153	65	371	5.7
lusearch	903	644	259	1547	5.92
pmd	3877790	3322241	558331	7200031	12.89
sunflow	3931888	3649527	285740	4026304	14.09
tomcat	133435	94416	38933	227851	5.85
xalan	7	4	3	14	1.33

Table 7.4: The number of events per benchmark for the `hasNext` property. The total number of events is equal to the number of hasNext events plus the number of next events as if a hasNext event is false the instrumentation filters it out and creates an end event. The number of end events gives the number of iterators.

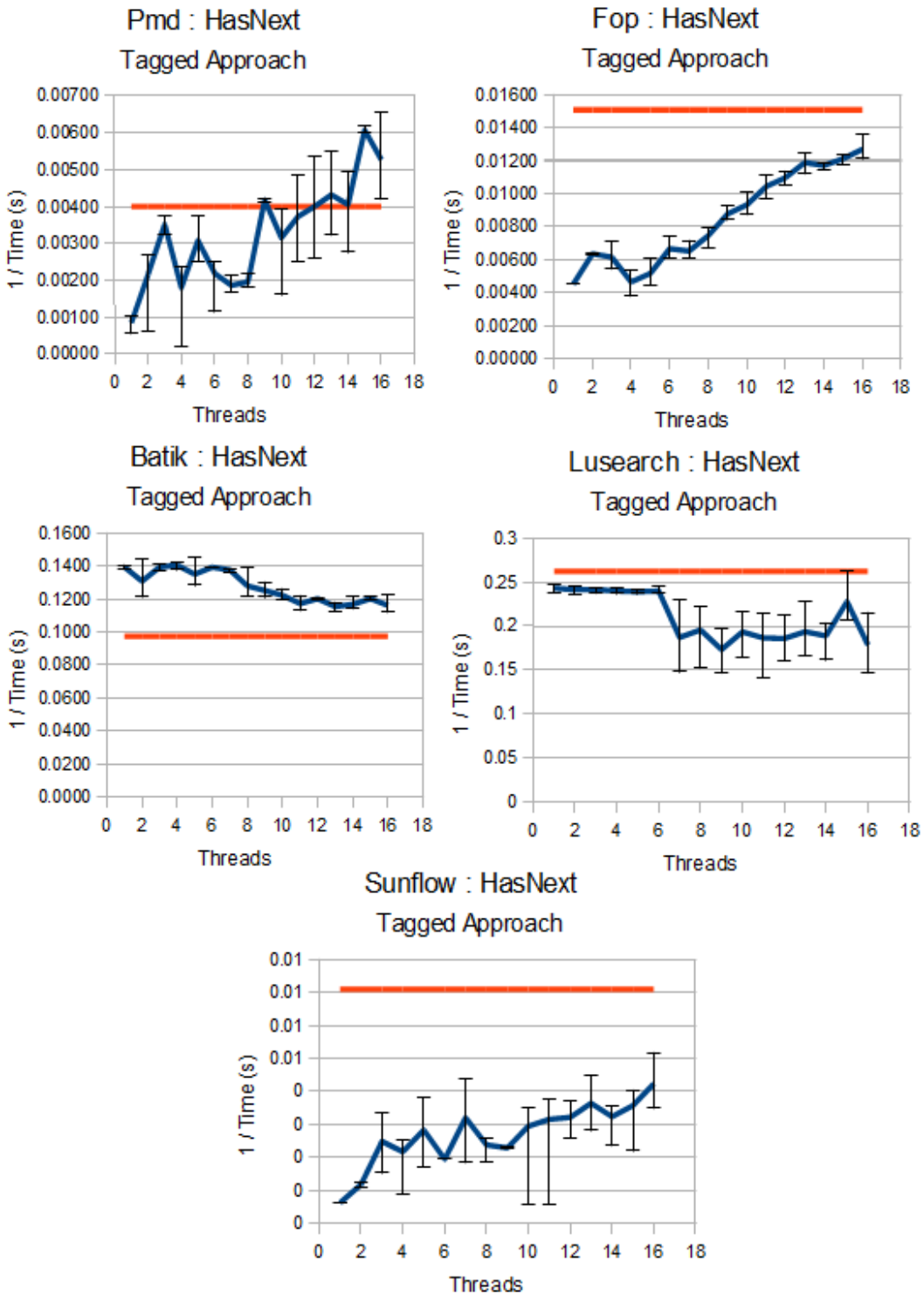


Figure 7.1: A look at the tagged approach for the `hasNext` property for selected benchmarks. The straight (orange) horizontal lines indicate the results for the clustered approach.

The alternative approaches applied to the avrora benchmark did not complete within a reasonable time, this was attributed to a combination of having almost twice as many events as any other benchmark and the monitoring process interfering with the intense synchronisation the workload carries out - recall that this benchmark simulates AVR mico-controllers and their communication.

Figure 7.1 shows the scalability for a selected number of benchmarks using the tagged approach. For the (multithreaded) pmd and (singe-threaded) fop benchmarks the approach scaled reasonably well - both of these workloads have a large number of events. In comparison the batik and lusearch benchmarks, which have considerably fewer events, ran slower when more threads were used.

Of all the benchmarks presented here only the pmd benchmark saw the tagged approach perform better than the clustered approach, and when considering the fluctuations this can only be confidently said for 16 threads.

For the lusearch property the fluctuations seen can be explained by the small number of iterators present (259) as this increases the effects of any imbalance caused by the naive hash function used. For the sunflow and pmd benchmarks the fluctuations may be a result of thread scheduling - the in monitor per-thread approach is used and therefore at least 16 monitor threads, if not more, were created.

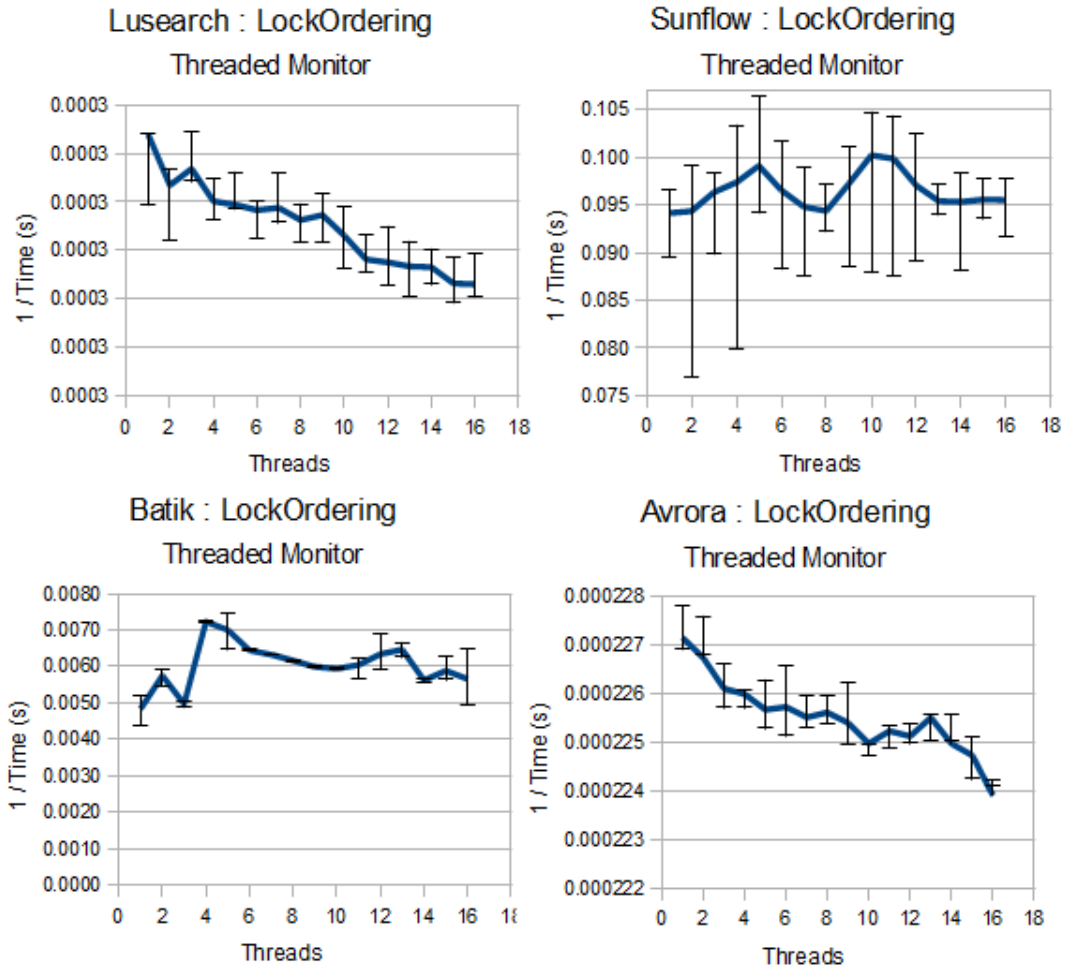
The shapes of these graphs are hard to interpret - more information on the distribution of iterations to iterators and iterators to monitors is required. The shape of the graph for the fop benchmark is what might be expected - a movement to two or three threads sees initial gains, the extra interference is felt when moving to four or five threads and then beyond this the gains outweigh the interference.

LockOrdering The number of active rule activations has the potential to grow large whilst monitoring this specification as the workloads use locking extensively and a term must remain in the frontier for as long as a lock is held. Therefore the parallel monitor is used. It should be noted that this specification only applies to multithreaded programs and that this specification will serialise the program. The Fop and Xalan benchmarks are not considered.

The results for monitoring this property are given for a selected number of benchmarks in Figure 7.2. Generally very little benefit was gained from running with multiple threads. The Sunflow and Batik benchmarks saw some scalability but this was not that great - translating to self-speedups of 1.06 and 1.49 respectively. Although as can be seen from the graphs the results for the sunflow benchmark fluctuate greatly and a maximum self-speedup of 1.22 was seen.

Figure 7.2 also gives the events and maximum and average single state sizes for each benchmark. From this it can be seen that the batik benchmark has by far the largest single state and this explains why this benchmark saw some, if not much, improvement.

The reason very little improvement is seen is that the amount of work done on each step is very small, making the amount of work carried out in parallel itself small and parallel overheads, such as load imbalance, large.



Benchmark	Events	Single State Sizes	
		Maximum	Average
Lusearch	6572830	602	192
Sunflow	2758	137	110
Batik	142430	2100	575
Avrora	9498794	501	179

Figure 7.2: The results of the `LockOrdering` property being monitored for a selected number of benchmarks.

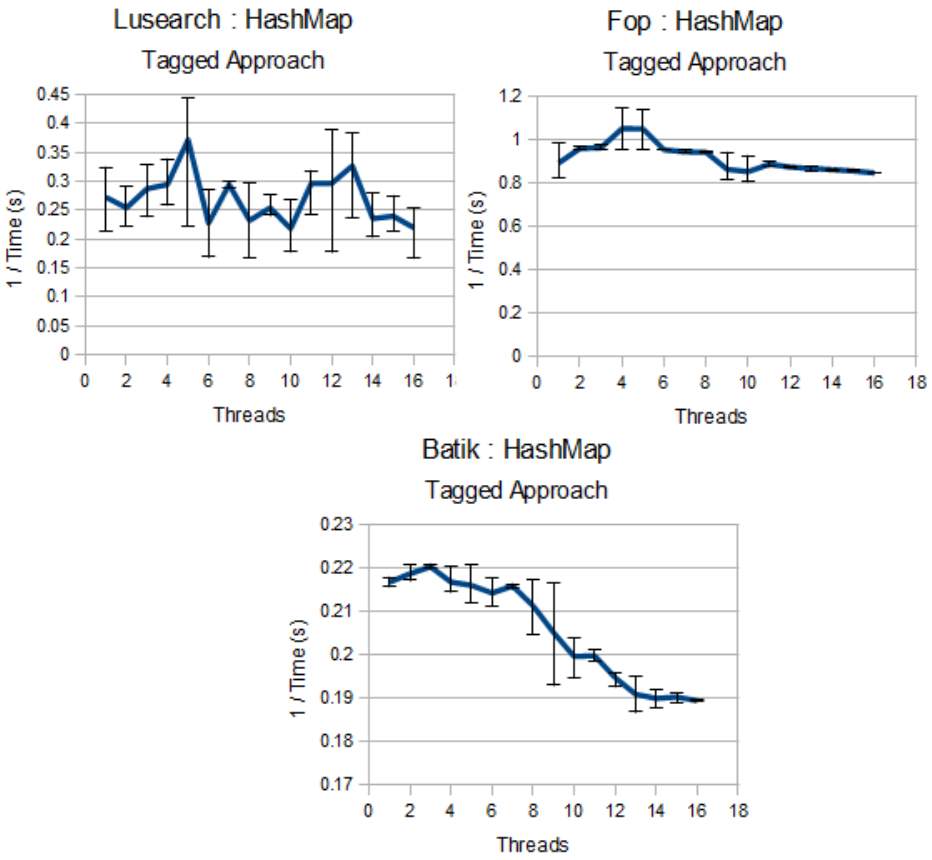


Figure 7.3: The results of the three benchmarks being monitored for the HashMap property using the tagged approach.

HashMap - Both the clustered and tagged approaches were taken here. Both approaches performed reasonably where applied although only in one or two cases were results for the clustered or tagged approaches better than those for the basic monitor approach. The best improvement found was for the Fop benchmark where the tagged approach performed 1.32 times faster than the basic monitor approach.

Figure 7.3 gives the scalability results for three of the benchmarks running the tagged approach on the HashMap property and the table below gives the number of each different event. It can be seen that only the batik benchmark could have failed. We are probably seeing a lot of memory contention here as the objects are being held by the monitor for as long as they live and in the batik benchmark the monitor thread is less likely to be running close to the program thread than in the lusearch benchmark, which might indicate why more consistent slowdown is seen for batik and why lusearch fluctuates more.

Benchmark	add	remove	contains	total
Lusearch	1625	0	0	1625
Fop	173	0	0	173
Batik	510	0	14	524

FailSafeIter and FailSafeEnum - These two similar specifications present either a clustered or non-blocking workload as an execution trace. However running with these approaches did not give favourable results where applied - for example with the Fop benchmark a slowdown of 93.53 was increased to 106.5 using the clustered approach and 157.16 using the non blocking approach.

SyncAll and SyncIteration and LeakingSync These specifications are rarely present in the benchmarks. No examined benchmarks presented the SyncIteration property and of the examined benchmarks only Batik presented the SyncAll property and a small number presented the LeakyingSync property.

For the LeakingSync property the clustered approach was applied, which ran at least twice as slow as the basic monitor approach, and at worst six times slower. The basic monitor approach saw between a 1.62 and 47.39 slowdown compared to unmonitored benchmark times.

7.1.4 Interference Results

This section briefly discusses the difference between the monitoring time and benchmark pass time for the *hasNext* property for two of the benchmarks. This difference indicates, to an extent, how much the benchmarks were interfered with.

Timings are given in Table 7.5 and it is clear that the nonblocking approaches interfere least with the Fop benchmark - the slowest run with a nonblocking approach is three times faster than the faster blocking approach and eight times faster than the slowest. The tagged approach allows the benchmark to pass in 10 seconds as opposed to 40 seconds when using the base monitor whilst the overall monitoring time is only just under 80 seconds compared to the monitoring time with the base monitor of just over 40 seconds.

For the Lusearch benchmark the story is a little different. Firstly the workload is a lot smaller and secondly the Lusearch benchmark is multithreaded. The additional monitor threads will interfere with the program's threads by competing for processor time.

	Bare	Monitor	Clustered	Tagged	NonBlocking	Post NonBlocking
Fop						
Pass	1346	40803	94562	10891	11293	11734
Monitor	-	41334	66347	78753	230638	235826
Slowdown	-	30.31	70.25	8.09	8.39	8.72
Lusearch						
Pass	2003	2368	4679	3504	2729	
Monitor	-	2635	5584	3802	3496	-
Slowdown	-	1.18	2.34	1.75	1.36	

Table 7.5: Pass and monitor Timings (in milliseconds) for the Fop and Lusearch benchmarks.

7.2 Summary

Monitoring the DaCapo benchmarks gave barely any good results - the number of times an approach using the concurrency offered by a multicore system was insignificant and these approaches often made the benchmarks run much slower than with just the base monitor.

Where the threaded monitor was used very minimal speedup was seen - a maximum of 1.49 self-speedup for the Batik benchmark saving roughly a minute for a three minute run. For multithreaded benchmarks no speedup was observed, in fact the monitoring process slowed down dramatically when a threaded monitor was used.

A limited study into interference was made and, as expected the nonblocking approaches saw the benchmarks passing more quickly than the blocking approaches - suggesting that they interfered less with the benchmarks. It should also be noted that throughout the course of the investigations none of the benchmarks failed (the DaCapo suite checks outputs) and so the monitoring process never noticeably altered the execution of a benchmark.

Chapter 8

Conclusion

This chapter first considers whether the aims and objectives of the project have been met, and then goes on to discuss conclusions that have been drawn about runtime monitoring and multicore machines and possible future work.

8.1 Have the Aims and Objectives Been Met?

In section 1.2 the following aims and objectives were given :

Aims

A Explore architectural and algorithmic ways to

i Increase the *performance* of, and

ii Decrease the *interference* of

the RuleR runtime monitoring tool through the use of multicore machines.

B Demonstrate scope for improvement through practical experimentation.

Objectives

1. Create a number of conceptual architectures and refine them through prototyping,
2. Implement these architectures within a coherent framework,
3. Evaluate these architectures for *efficiency* and *interference* through both microbenchmarks and performing runtime monitoring on a real-world program,

During this project the runtime monitoring tool RuleR was analysed and a number of different approaches to running the tool in parallel were developed. An experimental framework was implemented to include these approaches, as well as an optimised parallel RuleR monitor. Finally the framework and monitor were evaluated through a number of microbenchmarks and an application to a number of real-world applications from the DaCapo benchmark suite.

The majority of the evaluation focused on the *performance* of the framework and monitor, rather than the *interference*. However the microbenchmark discussed in section 6.9 showed that the timing properties monitorable by the framework and monitor were much reduced and section 7.1.4 showed that non-blocking approaches using the framework allows the monitored program to finish sooner.

The aims and objectives of this project were met and as a result some conclusions can be drawn about the applicability of the concurrency offered by multicore machines to the RuleR runtime verification tool, and some indication is given to how this might extend to the field of runtime monitoring in general.

8.2 Conclusions Drawn About Runtime Monitoring and Multicore Machines

Firstly, the concurrency offered by multicore systems can be used to increase the performance and decrease the interference of the rule-base runtime verification tool RuleR. However the kinds of workload where this is applicable are limited. The main conclusion of this project is that the 'runtime' element of runtime monitoring means that the work being done at any one point is too small to effectively parallelise. The majority of properties may be monitored by using a single state containing tens of terms with only one or two rules firing on each step. Further work should focus on increasing the amount of work available to the parallelisation techniques by grouping together pieces of work carried out at separate points in the monitoring process.

The taggable approach seemed the most promising as in the interleaved workload described in section 6.6.2 scalability was observed for a relatively simple property using a single state containing a handful of terms. However this scalability was only really achieved when events were interleaved and the number of iterators and work per iterator were very large.

Two areas that should have been explored but were left out due to time constraints were firstly the act of saving up events and evaluating them all in one step - to increase the amount of work done in a single step, and the effects of monitoring many different properties at the same time. These are not included within further work as they fall within the scope of this project.

To summarise the major lessons learnt were :

1. Improvements are more likely to be found in examining the work handed to the monitor rather than the monitor itself.

2. If simple properties are to be monitored the focus should be on reducing the size of the execution trace evaluated by a single monitor, as paralellising the internals of the monitor will see little benefit due to the small amount of work being carried out on each step.

3. For a rule-based monitor, such as RuleR, the amount of work being carried out on each step *per term* appears to be more important than the number of rules being evaluated when deciding whether the gains of parallelisation outweigh the overheads. Therefore there is only benefit in using a threaded monitor for complex properties that carry out a lot of work irregardless of the number of rule obligations they generate.

One major advantage of using a multicore machine for this project was the decreased communication overheads. A completely different approach would have been required if inter-thread communication was significantly higher, as would be in a cluster of workstations. This is especially relevant as the objects the monitor is operating over are still being used by the monitored application.

8.3 Future Work

This project has raised many questions and presented many possibilities for further work. In this section these are divided into possible extensions to the framework, future directions for the monitor and other avenues which could be explored.

8.3.1 Framework

There are a number of ways the framework could be extended both in general structure and in the architectures/components implemented within it.

Firstly there is no preprocessing carried out on the specification - this could be done to identify redundant events, dependent events and taggable independent traces. A redundant event is one that by its absence would never alter the outcome of the monitor. A dependent event is one that could never alter the outcome of the monitor unless the event it depends on has occurred. And a taggable independent trace is, as the name implies, is a sequence of events that can be uniquely identifiable by a single, or collection of, observations it refers to. By identifying these events or traces the framework could then selectively filter and decompose the execution trace as it receives it. Care would have to be taken to ensure that the `assert` set (one rule from this set must be present on every step of the monitor) is included in the preprocessing as this is a powerful tool that, sadly, can mean an event will always be required.

Currently the only message sending implementation is through a direct call on the monitor wrapper. This could be implemented in other ways, such as with Java Remote Method Invocation (RMI). This would allow the monitor wrapper and reply mechanism to exist in separate JVMs, potentially on different machines.

If the framework were to be used for reactive monitoring a rollback facility would be required - allowing the framework to undo steps taken by monitors. This would be more complicated than simply recording previous events - each monitor would need to record previous frontiers/single states. This could be done by saving changes made and undoing these in reverse order if rollback were required. The framework would also need to work out which monitors would need to be rolled back and how far - as the program would be going back to a particular point in its execution any of the monitors sent an event after this point would need to be rolled back. This would also involve scanning queues and removing obsolete events. To do all this events would need to be timestamped and some strict ordering of events would be required - hopefully the timestamps would suffice but if the monitored program was being ran on a distributed system where the individual system clocks were not synchronised another method would be required. Finally the user/instrumentation/running program itself would be expected to deal with rolling back the monitored program as this would require knowledge of what the program is doing and how it is operating.

8.3.2 Monitor

The main further extension that would most benefit the monitor is for it to be able to adapt to the execution trace it is monitoring. This is done very slightly in this project - for example monitoring is only

carried out in parallel if the frontier/single state is above a certain size and the specification is inspected to find out whether the property is deterministic or not. However there are many other parameters which are currently hard coded in or use a naive method to dynamically change them.

For example the cleaner thread should be able to learn how often a state needed to be cleaned and the monitor should be able to learn what the smallest size of frontier/single state would be worth monitoring in parallel for a particular run. The way a state was evaluated in parallel could also be altered depending on the kind of workload displayed in the execution trace.

Other extensions include a method for more quickly checking if a rule will fire. This could be done by statically examining the specification and constructing BDD (Binary Decision Diagram) of observations that *could* fire a rule - each observation could be passed through this to give a maximum pass set of rules that it could fire and only these would need to be checked. Other similar methods for reducing the number of rules checked could be constructed.

8.3.3 Other Avenues

One area not explored at all in this project is that of offline monitoring. This is a very different concept from online monitoring as the data to be operated over - the execution trace, is available in its entirety at the beginning of monitoring. This alleviates the main problem with parallelising online monitoring as the worksize is inherently many times larger.

A number of different approaches could be taken to using multicore systems to increase the performance of offline monitoring. The main requirement is to decompose the execution trace into a number of smaller traces whilst maintaining enough information to carry out monitoring. A few suggestions of how to do this are listed below

- Extract independent traces by examining the specification, as is done when using the taggable approach.
- Simply decompose the execution trace between available threads. The segments would each be passed forwards generating an *add* set for each segment. An *add* set would consist of a number of pairs of terms and would indicate that if the first term were present at the beginning of the segment the second would be present at the end. The *initials* set could be used to combine these *add* sets to give *present* sets for the beginnings of each segment. A *present* set would indicate which terms were present at the beginning of a segment. Then each thread could work through its segment using the *present* set to generate the value of that segment - this would then have to be combined. This approach would require each segment to be inspected twice so linear speedup is unlikely.

8.3.4 Summary

The question posed at the beginning of this dissertation - 'whether rule-based runtime verification can benefit from being used within a multicore system setting' has not been answered yet. This study shows that there is some promise and points in the direction of examining the work handed to the monitors rather than the monitors themselves.

This final section lays out some further ideas for how a multicore system setting could benefit the field of runtime monitoring mainly in the directions of bringing offline and online monitoring closer together and allowing the monitoring process to adapt to the execution trace being monitored.

Appendix A

Further RuleR Examples

A small number of extra examples to demonstrate the RuleR tool are given here. These examples present more advanced features of the tool including non-determinism and monitor combination.

A.1 Palindrome

- Here two approaches to detecting palindromes are given. The first approach is deterministic and demonstrates that rules are higher order, and therefore can be used to paramaterise other rules. The following specification checks for palindromes with a distinct midpoint. An example run of the specification is also given.

```
ruler Pal{
    observes add(int), mid,start,end;
    state Start : start -> Up(Down(end,End));
    state End : end -> Ok;
    state Up(r:rule){
            add(c:int) -> Up(Down(c,r));
            mid -> r;
            }
    state Down(c:int,r:rule) :  add(c) -> r;

    assert Up,Down,Start,End;
    succeed End;
    initials Start;
    forbidden Up,Down;
}
```

Obs	Frontier	Value
	{Start}	
start	{Up(End)}	still_false
add(1)	{Up(Down(1,End))}	still_false
add(2)	{Up(Down(2,Down(1,End)))}	still_false
mid	{Down(2,Down(1,End))}	still_false
add(2)	{Down(1,End)}	still_false
add(1)	{End}	still_false
end	{Ok}	true

This next example demonstrates non-determinism in RuleR specifications. The specification extends the previous one, note that it is no longer necessary to explictly give a mid point.

```
ruler Pal{
    observes add(int);
```

```
    state End : add(c:int) -> Fail;
    state Up(r:rule) : add(c:int) -> Up(Down(c,r)) | r | Down(c,r);
    state Down(c:int,r:rule): add(c) -> r;

    assert Up,Down;
    initials Up(End);
    forbidden Up,Down;

}
```

Note the use of **assert** and **forbidden** in the specification. The assert statement ensures that states consisting of Down and End rules die if they are not fired in the next step, an alternative would be replacing the body of Down with {{: `add(c) -> r; -> Fail;` :}}. The forbidden statement means that the frontier only evaluates to still_true if it consists of a state without an Up or Down, meaning a palindrome has been found. The following table gives an example of verifying 'abccba', note that 'a' is a palindrome.

	New Frontier	Signal
	$\{\{U(E)\}\}$	
a	$\{\{U(D(a,E))\},\{E\},\{D(a,E)\}\}$	still_true
b	$\{\{U(D(b,D(a,E)))\},\{D(b,D(a,E))\},\{D(a,E)\}\}$	still_false
c	$\{\{U(D(c,D(b,D(a,E))))\},\{D(c,D(b,D(a,E)))\},\{D(b,D(a,E))\}$	still_false
c	$\{\{U(D(c,D(c,D(b,D(a,E))))),\{D(c,D(c,D(b,D(a,E))))\},$	
	$\{D(c,D(b,D(a,E)))\},\{D(b,D(a,E))\}$	still_false
b	$\{\{U(D(b,D(c,D(c,D(b,D(a,E))))))\},\{D(b,D(c,D(c,D(b,D(a,E)))))\},$	
	$\{D(c,D(c,D(b,D(a,E))))\},$	
	$\{D(a,End)\}\}$	still_false
a	$\{\{U(D(a,D(b,D(c,D(c,D(b,D(a,E)))))))\},\{D(a,D(b,D(c,D(c,D(b,D(a,E))))))\},$	
	$\{D(b,D(c,D(c,D(b,D(a,E)))))\},\{End\}$	still_true

A.2 Countdown

- The previous example represents a liveness properties, something must eventually happen - in the previous example this was completing the palindrome. Here an example of a safety property is given, something that must always happen. Note that the property 'p must always happen' is equivalent to 'not p must never happen'. The following countdown specification checks that the received number is less than the previous number.

```
ruler countdown{
   observes count(int);

  state Start : count(x:int) -> Check(x);
  state Check(x:int){
             count(y:int){:
                         y<x -> Check(y);
                             -> Fail;
              :}
  }

  initials Start;
}
```

Observation	Frontier	Value
	{Start}	
count(10)	{Check(10)}	still_true
count(8)	{Check(8)}	still_true
count(5)	{Check(5)}	still_true
count(6)	{Fail}	false

A.3 Combining Monitors

- This example demonstrates monitor parameterisation and monitor combination. Figure A.1 shows the Ruler specification and four example runs to demonstrate it. The specification consists of two monitors chained together. The first, `reqgrant` , represents the property that every request by a user for an object will eventually be satisfied and that no user will make two requests in a row before receiving a reply. Formally represented by this property, in a fixed-point temporal logic,

$$\forall u : user, o : object,\ vx. \quad \texttt{req}(o_i,u_j) \quad \rightarrow \circ \quad \mu y. \quad \begin{array}{l} \texttt{req}(o_k,u_j), o_i \neq o_k \rightarrow \circ Fail, \\ \texttt{grant}(o_i,u_j) \rightarrow \circ x \\ \neg\texttt{grant}(o_i,u_j) \rightarrow \circ y \end{array}$$

$$\neg\texttt{req}(o_i,u_j) \quad \rightarrow \circ \quad Fail$$

The second, `users`, is paramterised by an integer x and represents the property that no user is granted more than x requests. This is acheived by RuleR's mechanism for combining monitors. In this case I have chained the two monitors together so that the outputs of the first monitor are sent as observations to the second monitor. The first monitor produces observations `granted(user)` whenever a user is granted a request, producing an observation event stream for the second monitor to monitor.

Three examples of incorrect behaviour detected by this example are given. Firstly an object is granted that was not requested. Secondly a user is granted too many requests. Thirdly a user makes a second request before being granted their first. Lastly an example of a correct trace is given, neither monitor can evaluate to true as they are both monitoring safety properties.

This example demonstrates how a property can be decomposed and managed by separate monitors. If we wanted to gather statistics on how many requests each user was being granted we would only need to change one of the monitors. If we wanted to do both, put a cap on granted requests and measure the number granted, we could implement a feature where the outputs of one monitor were forwarded to two different monitors. RuleR offers other methods for combining monitors, such as conditional monitors - $M_1?M_2 : M_3$.

```
ruler users(x:int,gin:obs){
   always N{gin(o:obj), !C(i:int,o) -> C(1,o);}
   state C(i:int,o:obj){
       gin(o){: i<=x -> C(i+1,o); -> Fail :}
   }
   initials N;
}

ruler reqgrant(gout:obs){
   observes req(obj,obj), grant(obj,obj);
   always R{ req(o:obj,u:obj), !W(x:obj,u) -> W(o,u);}
   state W(o:obj,u:obj){ grant(o,u) -> gout(u);}
   assert R,W;
   initials R;
   forbidden W;
   outputs gout;
}
monitor {
  uses R : reqgrant, U : users;
  locals granted(obj);
  run (R(granted) >> U(2,granted)).
}
```

Observation	FrontierR	ValueR	FrontierU	ValueU
	{R}		{N}	
req(obj1,user1)	{W(obj1,user1),R}	still_false	{N}	still_true
req(obj2,user1)	{W(obj2,user1),R}	still_false	{N}	still_true
grant(obj1,user1)	{R}	still_false	{C(1,user1),N}	still_true
req(obj2,user3)	{W(obj2,user1),W(obj2,user3),R}	still_false	{C(1,user1),N}	still_true
grant(obj1,user1)	{W(obj2,user1),W(obj2,user3),R}	false	{C(1,user1),N}	still_true
	{R}		{N}	
req(obj1,user1))	{W(obj1,user1),R}	still_false	{N}	still_true
grant(obj1,user1))	{R}	still_true	{N}	still_true
req(obj1,user1))	{W(obj1,user1),R}	still_false	{C(1,user1),N}	still_true
grant(obj1,user1))	{R}	still_true	{C(1,user1),N}	still_true
req(obj1,user1))	{W(obj1,user1),R}	still_false	{C(2,user1),N}	still_true
grant(obj1,user1))	{R}	still_true	{C(2,user1),N}	still_true
req(obj1,user1))	{W(obj1,user1),R}	still_false	{C(2,user1),N}	still_true
grant(obj1,user1))	{R}	still_true	{C(2,user1),N}	false
	{R}		{N}	
req(obj1,user1))	{W(obj1,user1),R}	still_false	{N}	still_true
req(obj2,user1))	{W(obj1,user1),R}	false	{N}	still_true
	{R}		{N}	
req(obj1,user1))	{W(obj1,user1),R}	still_false	{N}	still_true
grant(obj1,user1))	{R}	still_true	{C(1,user1),N}	still_true

Figure A.1: User grant request example

Appendix B

Specifications

The following two sections give the specifications used in the experimental and evaluation chapters and a table is provided here to summarise these.

Name	Relevant Sections	Page
Workload 1	6.3	92
Workload 2	6.3	92
Workload 3	6.3	92
Workload 4	6.3 and 6.2	92
Workload 1ND	6.4	93
Workload 2ND	6.4	93
Workload 3ND	6.4	93
Workload 4ND	6.4	93
Interleaved	6.6.2	94
End Iteration	6.5, 6.6, 6.7, 6.8 and 7.1.2	94
Lazy Iteration	6.8	94
Set Iteration	6.8	96
Original Iteration	6.8	96
Timing	6.9	95
SyncAll	7.1.2	96
SyncIteration	7.1.2	96
FailSafeEnum/Iter	7.1.2	96
HashMap	7.1.2	96
LeakingSync	7.1.2	97
LockOrdering	7.1.2	97

B.1 From Experimental Chapter

B.1.1 Workloads 1-4

From section 6.3 , workload 4 is also used in section 6.2.

Workload 1

```
ruler Huge{
 observes a(obj), b, c;
    always A{ a(i:obj) -> C(i),D(1,i); }
    state B{ b -> B;}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:obj){ c -> D(i+1,o); }
    assert A , B, D;
    initials  A, B;
}

monitor {
 uses M: Huge;
 run M .
}
```

Workload 2

```
ruler Huge{
 observes a(obj), b, c;
    always A{ a(i:obj) -> C(i),D(1,i); }
    state B{ b -> B;}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:obj){  b -> D(i+1,o); }
    assert A , B, D;
    initials  A, B;
}

monitor {
 uses M: Huge;
 run M .
}
```

Workload 3

```
ruler Huge{
 observes a(obj), b, c;
    always A{ a(i:obj) -> C(i),D(1,i); }
    state B{ b -> B;}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:obj){  A,B -> D(i+1,o); }
    assert A , B, D;
    initials  A, B;
}

monitor {
 uses M: Huge;
 run M .
}
```

Workload 4

```
ruler Huge{
 observes a(obj), b, c;
    always A{ a(i:obj) -> C(i),D(1,i); }
    state B{ b -> B;}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:obj){
                        A,B,C(o) -> D(i+1,o);
                        !A -> A;
                        !B -> B;
                        !D(i,o) -> D(i,o);
    }
    assert A , B, D;
    initials  A, B;
}

monitor {
 uses M: Huge;
 run M .
}
```

B.1.2 Workloads 1ND-4ND

From section 6.4 - an attempt at non deterministic versions of the above .

Workload 1ND

```
ruler Huge{
 observes a(obj,int), b(int), c;
    always A{ a(o:obj,i:int) ->
       A,B(i),C(o),D(1,i) |
       A,B(i+1),C(o),D(1,i); }
    state B(i:int){ b(i) -> B(i);}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:int){  c -> D(i+1,o); }
    assert A , B, D;
    initials  A;
}

monitor {
 uses M: Huge;
 run M .
}
```

Workload 2ND

```
ruler Huge{
 observes a(obj,int), b(int), c;
    always A{ a(o:obj,i:int) ->
       A,B(i),C(o),D(1,i) |
       A,B(i+1),C(o),D(1,i); }
      state B(i:int){ b(i) -> B(i);}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:int){  b -> D(i+1,o); }
    assert A , B, D;
    initials  A;
}

monitor {
 uses M: Huge;
 run M .
}
```

Workload 3ND

```
ruler Huge{
 observes a(obj,int), b(int), c;
    always A{ a(o:obj,i:int) ->
       A,B(i),C(o),D(1,i) |
       A,B(i+1),C(o),D(1,i); }
    state B(i:int){ b(i) -> B(i);}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:int){  A,B -> D(i+1,o); }
    assert A , B, D;
    initials  A;
}

monitor {
 uses M: Huge;
 run M .
}
```

Workload 4ND

```
ruler Huge{
 observes a(obj,int), b(int), c;
    always A{ a(o:obj,i:int) ->
       A,B(i),C(o),D(1,o) |
       A,B(i+1),C(o),D(1,o); }
    state B(i:int){ b(i) -> B(i);}
    always C(x:obj){ c -> Ok; }
    state D(i:int,o:obj){
                      A,B(j:int),C(o) -> D(i+1,o);
                      !A -> A;
                      !C(o) -> C(o);
                      !D(i,o) -> D(i,o);
    }
    assert A , B, D;
    initials  A;
}

monitor {
 uses M: Huge;
 run M .
}
```

B.1.3 Interleaved specification

From section 6.6.2.

Interleaved

```
ruler Tag{
 observes update(obj);
    state S{   update(o:obj), !C(o,x:int) -> C(o,1);   }
    state C(o:obj,i:int){ update(o) -> C(o,i+1);  }
   initials  S;
}

monitor {
 uses M: Tag;
 run M .
}
```

B.1.4 Different `hasNext` specifications

These are used throughout project.

End Iteration

```
ruler EndIteration{

 observes  hasNext(obj), next(obj), end(obj);

    always Start {
      hasNext(i:obj) -> Next(i);
      end(i:obj) -> !Next(i);
    }

    state Next(i:obj) {
      next(i) -> Ok;
    }

    assert Start, Next;
    initials  Start;
}

monitor {
 uses M: EndIteration;
 run M .
}
```

Lazy Iteration

```
ruler LazyIteratorMonitor{

 observes  hasNext(obj), next(obj);

    always Start {
      hasNext(i:obj) -> Next(i);
    }

    step Next(i:obj) {
      next(i) -> Ok;
    }

    assert Start, Next;
    initials  Start;
}

monitor {
 uses M: LazyIteratorMonitor;
 run M .
}
```

Set Iteration

```
ruler SetIteration{
 observes  hasNext(obj), next(obj);

    always Start(x:set) {
      hasNext(i:obj), !Next(i) -> Next(i);
      hasNext(i:obj), !Next(i), Next(j:obj){|
             x += { Next(j) } -> Ok;
             !Next(j);
          |}
      next(i:obj), !Next(i), Next(i) in x -> Ok;
    }

    state Next(i:obj) {  next(i) -> Ok;  }
    assert Start, Next;
    initials  Start({ });
}

monitor {
 uses M: SetIteration;
 run M .
}
```

Original Iteration

```
ruler HasNext{

 observes  hasNext(obj), next(obj);

    always Start {
        hasNext(i:obj) -> Next(i); }

    state Next(i:obj) { next(i) -> Ok; }

    assert Start, Next;
    initials  Start;
}

monitor {
 uses M: HasNext;
 run M .
}
```

B.1.5 Timing specification

From section 6.9.

Timing

```
ruler Timing{
 observes grow(int), start(long), stop, time(long);
    always S{ start(l:long), time(t:long) -> W(l,t);  }

    state W(l:long,t:long){
      stop, time(n:long) {:  n < (l+t) -> Ok;   default -> Fail;   :}
    }

    always G{ grow(i:int) -> A(i); }
    state A(i:int){ !G -> Fail;  }
    initials  S, G;
}

monitor {
 uses M: Timing;
 run M .
}
```

B.2 From Evaluation Chapter

These specifications are derived from the those given in [20] and [58].

SyncAll

```
ruler SyncContainsAll{
 observes  create(obj), update(obj,obj);
    always Start {
      create(t:obj) -> Hold(t);
      update(t:obj,a:obj), !Hold(t) -> Ok;
    }

    always Hold(t:obj){  update(t,t) -> Ok;}

    assert Start, Hold;
    initials  Start;
}

monitor {
 uses M: SyncContainsAll;
 run M .
}
```

FailSafeIter/Enum

```
ruler FailSafeEnum{
 observes  create(obj,obj), next(obj),
              update(obj);

    always Start() {
       create(ds:obj,e:obj) -> Update(ds,e);
    state Update(ds:obj,e:obj){
       update(ds) -> NotNext(e);   }
    state NotNext(e:obj){
      next(e) -> Fail;  }

    initials  Start;
}

monitor {
 uses M: FailSafeEnum;
 run M .
}
```

SyncIteration

```
ruler AsyncIteration{
      observes create(obj), async(obj);

      always Start(){
            create(c:obj) -> Hold(c);
                  async(c:obj), !Hold(c) -> Ok;
      }

      always Hold(c:obj){ async(c) -> Fail;}
      assert Start,Hold;
      initials Start;
}

monitor{
uses M : AsyncIteration;
run M .
}
```

LeakingSync

```
ruler LeakingSync{
 observes  create(obj), update(obj),blank;

    always Start {
      create(o:obj) -> Check(o);
      update(o:obj), !Check(o) -> Sig();
    }

    always Check(o:obj){ update(o) -> Fail;}
    always Sig(){  blank -> Fail; }
    assert Start;
    initials  Start;
}

monitor {
 uses M: LeakingSync;
 run M .
}
```

HashMap

```
ruler Left{
    observes add(obj,obj,int),
      remove(obj,obj,int), cont(obj,obj,int);

      always Start(){
           add(m:obj,o:obj,h:int) -> Remove(m,o,h);
           cont(m:obj,o:obj,h:int) -> Ok;
      }
      state Remove(m:obj,o:obj,h:int){
            remove(m,o,i:int){:
                    i==h ->Ok;
                    default -> Fail;
                 :}
      }
      assert Start, Remove;
      initials Start;
}

ruler Right{
      observes add(obj,obj,int),
         remove(obj,obj,int), cont(obj,obj,int);

      always Start(){
         add(m:obj,o:obj,h:int) -> Contains(m,o,h);
         remove(m:obj,o:obj,h:int),
            Contains(m,o,h) -> !Contains(m,o,h);
      }
      state Contains(m:obj,o:obj,h:int){
            cont(m,o,i:int){:
                    i==h -> Contains(m,o,h);
                    default -> Fail;
                 :}
      }
      assert Start, Contains;
      initials Start;
}

monitor{
   uses L : Left, R : Right;
   run (L & R) .
}
```

LockOrdering

```
ruler LockOrdering{

 observes  lock(obj,obj), unlock(obj,obj);

    always Start(){
        lock(t:obj,l:obj) -> One(t,l);
   unlock(t:obj,l:obj) -> Four(t,l);
    }
    step One(t:obj,l:obj){
       !unlock(t,l) -> One(t,l);
  lock(t,l2:obj) -> Two(l,l2);
    }
    always Two(l1:obj,l2:obj){
        lock(t:obj,l2) -> Three(t,l1,l2);
    }
    step Three(t:obj,l1:obj,l2:obj){
       !lock(t,l1) -> Three(t,l1,l2);
  unlock(t,l2) -> Ok;
    }
    state Four(t:obj,l:obj){
       unlock(t,l) -> Ok;
  lock(t,l2:obj) -> Four(t,l);
    }

    initials Start;
    forbidden One;

}

monitor {
 uses M: LockOrdering;
 run M .
}
```

Bibliography

[1] http://csd.informatik.uni-oldenburg.de/~jass/.

[2] http://dacapobench.org.

[3] http://maude.cs.uiuc.edu/.

[4] http://runtime-verification.org/.

[5] http://spinroot.com/.

[6] http://www.eclipse.org/aspectj/.

[7] http://www.spass-prover.org/.

[8] J. Abrial, S.A. Schuman, and B. Meyer. A specification language. *On the Construction of Programs*, 1980.

[9] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[10] J. Barnat, L. Brim, and Petr Ročkai. Scalable multi-core ltl model-checking. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 187–203, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] H. Barringer, M. Fisher, D.M. Gabbay, G. Gough, and R. Owens. Metatem: An introduction. *Formal Asp. Comput.*, 7(5):533–549, 1995.

[12] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[13] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 2010.

[14] H. Barringer, K. Havelund, D.E. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. *Runtime Verification, 9th International Workshop, RV 2009*, pages 1–24, 2009.

[15] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J Logic Computation*, November 2008.

[16] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - java with assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[17] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? *7th International Workshop, RV 2007*, pages 126–138, 2007.

[18] K. Beck. *Test-Driven Development By Example*. Addison-Wesley, 2003.

[19] H. Belhaouari and F. Peschanski. A lightweight container architecture for runtime verification. *8th International Workshop, RV 2008*, pages 173–187, 2008.

[20] Eric Bodden and Laurie Hendren. A staged static program analysis to improve the performance of runtime monitoring. In *In Ernst [8*, pages 525–549. Springer, 2007.

[21] F. Chen, M. d'Amorim, and G. Rosu. Checking and correcting behaviors of java programs at runtime with java-mop. *Electr. Notes Theor. Comput. Sci.*, 144(4):3–20, 2006.

[22] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 2004.

[23] E. D. Dijkstra. A constructive approach to the problem of program correctness. circulated privately, August 1967.

[24] D. Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[25] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[26] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82*, pages 169–180, New York, NY, USA, 1982. ACM.

[27] T.G. Evans and D.L Darley. On-line debugging techniques: a survey. In *AFIPS '66 (Fall): Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 37–50, New York, NY, USA, 1966. ACM.

[28] R.W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967.

[29] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[30] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, 1988.

[31] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[32] J. Goubault-Larrecq and J. Olivain. A smell of orchids. *Runtime Verification: 8th International Workshop, RV 2008*, pages 1–20, 2008.

[33] K. Havelund and G. Rosu. Java pathexplorer - a runtime verification tool. In *In The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, 2001.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[35] C. A. R. Hoare. Retrospective: an axiomatic basis for computer programming. *Commun. ACM*, 52(10):30–32, 2009.

[36] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[37] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Trans. Softw. Eng.*, 33(10):659–674, 2007.

[38] Cornelia P. Inggs and Howard Barringer. Ctl* model checking on a shared-memory architecture. *Form. Methods Syst. Des.*, 29(2):135–155, 2006.

[39] K. Kähkönen, J. Lampinen, K. Heljanko, and I. Niemelä. The lime interface specification language and runtime monitoring tool. *9th International Workshop, RV 2009*, pages 93–100, 2009.

[40] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[41] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language euclid. *SIGPLAN Not.*, 12(2):1–79, 1977.

[42] Doug Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[43] G.T. Leavens and Y. Cheon. Design by contract with jml. *unpublished*, 2004.

[44] I. Lee, H. Ben-abdallah, S. Kannan, M. Kim, I. Sokolsky, and M. Viswanathan. A monitoring and checking framework for run-time correctness assurance. In *In Proceedings of the 1998 Korea-U.S. Technical Conference on Strategic Technologies*, 1998.

[45] J. Mccarthy, , and J. Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[46] J. Mccarthy. Checking mathematical proofs by computer. *Proceedings Symposium on Recursive Function Theory*, 1962.

[47] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[48] B. Meyer, J.M. Nerson, and M. Matsuo. Eiffel: object-oriented design for software engineering. In *Proc. of the 1st European Software Engineering Conference on ESEC '87*, pages 221–229, London, UK, 1987. Springer-Verlag.

[49] S. Owre, J.M. Rushby, and N. Shankar. Pvs: A prototype verification system. *Automated DeductionCADE-11*, 1992.

[50] L.C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1987.

[51] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5, 1989.

[52] B. Plattner. *Monitoring Program Execution*. PhD thesis, Swiss Federal Institute of Technology Zrich, 1983.

[53] B. Plattner and J. Nievergelt. Special feature: Monitoring program execution: A survey. *Computer*, 14(11):76–93, 1981.

[54] A. Pnueli. The temporal logic of programs. In *SFCS '77*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[55] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer, 1998.

[56] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, 2002.

[57] H. Shin, Y. Endoh, and Y. Kataoka. Arve: Aspect-oriented runtime verification environment. *Runtime Verification, 7th International Workshop, RV 2007*, pages 87–96, 2007.

[58] Volker Stolz, Von Der Fakultt Fr Mathematik, Informatik Und, Naturwissenschaften Der Rheinisch-westflischen, Technischen Hochschule, Aachen Erlangung, Volker Stolz, Berichter Prof, Dr. Klaus Indermark, Prof Bernd Finkbeiner, Ph. D, and Dipl. inform Volker Stolz. Temporal assertions for sequential and concurrent programs, 2007.

[59] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[60] A. Turing. Checking a large routine. *The early British computer conferences*, pages 70–72, 1989.

[61] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.

[62] P. Wegner. The vienna definition language. *ACM Comput. Surv.*, 4(1):5–63, 1972.

[63] K. Zee, V. Kuncak, M. Taylor, and M.C. Rinard. Runtime checking for program verification. *Runtime Verification, 7th International Workshop, RV 2007*, pages 202–213, 2007.

[64] W. Zhou, O. Sokolsky, B.T. Loo, and I. Lee. Dmac: Distributed monitoring and checking. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 184–201. Springer, 2009.