MAIL - An Interaction Layer For Exploring The Use Of Multicore in Runtime Monitoring

Giles Reger, David Rydeheard and Howard Barringer

School of Computer Science, University of Manchester, UK
{giles.reger,david.rydeheard,howard.barringer}@cs.man.ac.uk

Abstract. This paper examines whether the concurrency offered by multicore systems can be utilised in runtime monitoring. We develop an interaction layer, MAIL, to experiment with different approaches to parallelising the interaction between monitor and application. Our experiments use the RULER runtime monitoring tool with special microbenchmarks and the DaCapo benchmark suite to show that multicore systems can reduce the overhead of runtime monitoring.

1 Introduction

With multicore chips finding their way into current desktop and laptop computers it is important to consider how the concurrency they offer can be harnessed in different areas of computer science. This paper reports on a study which investigated how multicore systems may be used to i) improve the performance of runtime monitors, and ii) reduce their associated overheads.

In general, a monitoring process introduces some form of overhead in terms of additional computation and communication. This potentially decreases usability and introduces undesired interference and changes of behaviour. Previous approaches to reducing (or eliminating) this overhead include *Offline Analysis* [4], where execution traces are recorded as log files and analysed offline, keeping runtime overhead and interference to a minimum, and *Partial Evaluation* [9], where static analysis techniques are employed at compile time to partially evaluate monitors. This study focusses on parallelism, specifically that available through multicore systems. To provide a flexible and general-purpose experimental platform for investigating parallelism in the runtime monitoring process we develop an interaction layer between the monitor and monitored application, MAIL (Monitor Application Interaction Layer). This allows us to implement various approaches and evaluate them for special microbenchmarks and the DaCapo benchmark suite [8]. Using this platform we found that:

- There are two approaches to parallelising the interaction between monitor and application: desynchronising interaction and partitioning the trace between different monitors running in parallel,
- The MAIL interaction layer can be used to achieve speed-ups for workloads we would expect to see in the runtime monitoring process, in some cases super-linear speedup is observed,

– Whilst some performance improvements can be achieved for the DaCapo benchmark suite, the benchmarks did not exhibit the expected workloads. Only 59 out of 179 experiments achieved an improvement in performance.

We draw the conclusion that multicore systems can be used to improve the performance of runtime monitors and reduce their associated overheads. However, we note that (i) our MAIL interaction layer sacrifices some performance gains for generality, and (ii) more work has to be done to understand the data-parallel nature of workloads that can be exhibited during the runtime monitoring process. We discuss these issues further later.

In Section 2, we consider runtime monitoring and the potential for parallelising this process, focussing on the interaction between monitor and application. MAIL is introduced in Section 3 and our experiments discussed in Section 4.

2 Potential for Parallelism

Formal approaches to runtime monitoring have been considered for over a decade [1, 7, 10, 11, 13–18, 20]. One approach has been to construct general purpose frameworks such as EAGLE [3], RULER [7, 6], LOGSCOPE [4] and TRACECONTRACT [5], all of which are essentially rule-based with the ability to handle parameterised data. This represents a very expressive approach with data parameterisation allowing memory to be encoded in rules, thus allowing the specification of context-free properties and beyond.

A rule-based monitor with data parameterisation typically uses rules of the form $ruleName(parameters) : preconditions \rightarrow obligations$, where preconditions and obligations can make use of the rule's parameters. A *rule instance* is an instantiation of a rule at runtime with actual parameter values. The monitor consists of a set of rule instances which is updated on receiving an event by discharging all rule instances with satisfied preconditions. Rules can be tagged with different lifetimes, from fully persistent to single-step.

We have used the RULER system [7, 6], and Java implementation, as a vehicle for this study, but the approach in this paper is general and could be applied to any runtime monitoring tool that supports events to be dispatched and feedback returned on a step-by-step basis.

In previous work [19], we attempted to parallelise the way that a RULER monitor handles events and found that the amount of work required to evaluate a single event was generally not large enough to benefit from parallelisation. For runtime monitoring as usually envisaged, with simple events and event processing, we believe this to be generally the case when considering a single step of a monitor. Here we focus on a different role for parallelism : parallelising the interaction between monitor and application.

Figure 1 illustrates the *normal* way in which a monitor and application interact. The monitor is synthesised from a specification and an instrumented application dispatches events to it, receiving feedback in return, which can be used to stop or alter the application's execution. The sequence of events sent to

the monitor is a *trace*. Where an application has more then one thread of execution (i.e. is multi-threaded), events created by different threads of execution are arbitrarily interleaved in the trace. Additionally, if a thread of execution must wait for the monitor to reply, these threads may synchronise on the monitor.

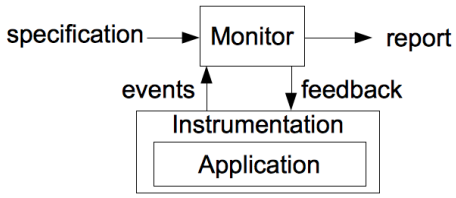

Fig. 1. Monitoring an Application

A monitor operating *online* will normally interact *synchronously*, meaning that the application must pause for a reply. Therefore, from the monitored system's point of view, the monitoring overhead consists of sending an event to the monitor and then waiting for the reply - it is the time waiting for the monitor to reply that has to be tackled. We can either attempt to *cover* this overhead by (partially) desynchronising the monitor and application, running the two processes in parallel, or we can *reduce* the overhead by a "data-parallel" approach that partitions the trace between separate monitors running in parallel.

We could completely desynchronise interaction, using a buffer for feedback or having no feedback at all. However, runtime monitoring is often used to detect undesired behaviour and to halt or alter the behaviour of the application if this is detected, meaning that in some cases synchronisation is required. A compromise can be found if we synchronise on only those events that can potentially lead to the specification failing, we call these events *fail-possible events*. We consider five types of synchronisation:

1. Synchronise on every event (*Normal Monitoring*)
2. Synchronise only on fail-possible events (*Clustered*)
3. Synchronise on the last event in a trace (*Asynchronous*)
4. Run monitoring online after application has run (*Post-Processing*)
5. Run monitoring offline after application has run (*Log File Analysis*)

Only the first three approaches are suitable for *continuous* monitoring, which does not assume a finite trace. Fully synchronous interaction implies that the monitor may run in the same thread as the application, but any level of asynchronous interaction requires the monitor to run in a separate thread. Fail-possible events can either be given by the user or inferred from the specification. The Clustered approach is a general case of the Normal and Asynchronous approaches and is so-called as we can view the events between fail-possible events as clusters. In Section 4, we evaluate the Clustered, Asynchronous and Post-Processing approaches only.

Desynchronising interaction only covers overhead, giving a theoretical limit to the amount that it can be reduced. In the Asynchronous approach almost all event processing overhead may be eliminated (except that monitoring may run longer than the application), however, the magnitude of overhead covered by the Clustered approach will depend on the time between fail-possible events and the time taken to process a single event. Therefore, where there are many

synchronisation points the average time to process a single event will have a large influence on the level of overhead. This can be reduced by partitioning the trace into subtraces to be processed by separate monitors running in parallel. To partition a trace we need to 'tag' each event with a reference to the monitor(s) it should be processed by. To do this we consider different *tagging schemes*:

- *Per Application Thread* - Different tags are given to events produced by different application threads,
- *Per Object* - Different tags are given to events related to different (sets of) object instances whose behaviour is separate in the specification,
- *Arbitrary* - Different tags are assigned according to some user-defined property relating events and their parameters separating different behaviours in the specification.

A partitioning of the trace could assign more than one tag to an event and is based on separating events that will not interfere with each other in the monitor.

Previous work in the area of runtime monitoring has applied some of these techniques, but not in the context of parallelism. Chen and Roşu developed a tool called `PMon`[11] that explores parametric trace slicing, which relates to 'per-object' partitioning here, and the RV System [18] also employs this trace slicing. Tracematches [1] is based on the AspectJ compiler and allows per-application-thread trace partitioning. LOGSCOPE [4] performs offline log-file analysis and Colombo et al. [12] use asynchronous monitoring to implement a compensation-aware runtime monitoring architecture in a tool called `cLARVA`.

This section discussed the two techniques we use to parallelise the interaction between monitor and application. The next section presents MAIL, an interaction layer used to explore implementations of these techniques.

3 MAIL: Monitor Application Interaction Layer

MAIL is a layer used to explore techniques for desynchronising interaction and the partitioning of traces. We first introduce the structure of MAIL and then the implemented architectures.

3.1 Structure

Figure 2 gives the structure of MAIL, showing the components used to pass events and feedback through the layer between monitor and application. Events are dispatched to the REPLY MECHANISM component where they are converted into *messages* and tagged. The message is passed to the MESSAGE SENDING component which first filters the message and sends the message to the relevant monitor if it exists, if not a new monitor is created. The MESSAGE FILTERING component allows for the removal and updating of messages, and the MONITOR CREATION component is a factory object for monitors. Lastly, the MONITOR WRAPPER component wraps up the monitor to allow different forms of communication with it, for example running the monitor in a separate thread. The

MONITOR WRAPPER dispatches the event to the monitor and passes the reply to the MESSAGE SENDING component, where it is filtered. The reply is then sent to the REPLY MECHANISM component where it is either returned directly to the instrumented application or buffered for later retrieval.

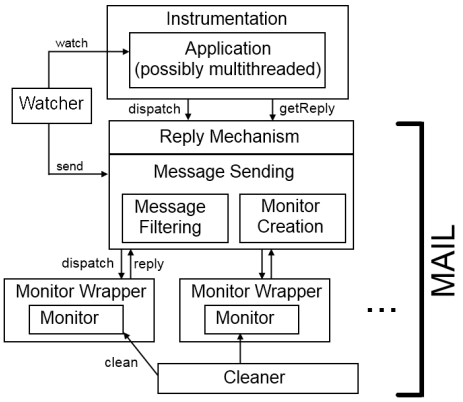

Fig. 2. Structure of MAIL

Special messages are used to create and end monitors, as well as start and stop them if running in separate threads. The MESSAGE FILTERING component can also be used to implement other special messages. There are two additional helper threads involved in MAIL that run in parallel with the application and monitors to perform additional helpful functions. The WATCHER thread is used when the application is multithreaded and sends a special message to the MESSAGE SENDING component when an application thread ends. Book-keeping and maintenance tasks within the monitor are factored out into a CLEANER thread.

MAIL is a general-purpose experimental platform in the sense that it is compatible with any monitoring tool that operates on a step-by-step basis. To use a different monitor we only need to reimplement the MONITOR WRAPPER component and CLEANER helper thread. In this investigation we use RULER monitors. One feature of RULER is that rule instances refer to monitored objects using *weak references*, so that when these objects go out of scope in the application these rule instances can be labelled as garbage and removed. As the amount of time a RULER monitor takes to process an event is dependent on the number of rule instances this feature is important for efficiency. However, so that errors can be reported, and further action taken if the monitoring is interactive, messages refer to monitored objects with *strong references*. This breaks the garbage rule instance removal process as these cannot be removed until that message has been cleared in the REPLY MECHANISM. As a result garbage rule instances can last longer in the monitor, potentially causing longer event processing times. We use the CLEANER thread to remove garbage rule instances.

To explore different approaches, several implementations of these components were constructed and combined - we call these combinations *architectures*.

3.2 Architectures

All architectures extend a single super-architecture object that provides a method for halting monitoring, this blocks and waits for the final reply from the monitor(s). Table 1 describes a subset of the implementations we constructed for the components. These are combined to create architectures and those evaluated in

Reply Mechanism	
Sync	After sending the message the thread waits for the reply
Async	When a message is sent the thread returns immediately and a `getReply` method can query the component for the reply
Partial	A no-wait dispatch sends the message, increments a counter and returns. Every reply decrements the counter. A wait dispatch sends the message and returns the reply when the counter reaches zero
Tag-Async	The same as *Async* but messages are tagged
Tag-Partial	The same as *Partial* with tagged messages and a counter per-tag
Message Sending	
Single	Only a single monitor is created at the start and message tags are ignored
Multiple	Uses a maximum of n monitors. A hash of the message tag into the range [0,n] is used to select (or create) the monitor to send a message to. Monitors are created on-demand.
Message Filtering	
Base	Deals with special messages such as monitor creation
Latest	Keeps track of the latest reply and returns it immediately
Halting	Halts the monitor and application as soon as a violation is detected
Monitor Creation	
RULER	Creates RULER monitors based on options passed by the instrumentation
Monitor Wrapper	
In-Thread	A RULER monitor runs in its own thread with communication via a buffer

Table 1. A description of the different component implementations

Architecture	Communication	Partitioning	Reply Mechanism	Additional Threads
Clustered	clustered	per-thread	*Tag-Partial*	$m + 2$
Tagged	asynchronous	arbitrary	*Tag-Async*	$n + 2$
Asynchronous	asynchronous	per-thread	*Tag-Async*	$m + 2$
Post-Processing	post-processing	per-thread	*Tag-Async*	$m + 2$

Table 2. A summary of the different architectures. The last column indicates the *additional threads* used where there are m threads in the monitored system, n monitors to be used and two helper threads.

the next section are summarised in Table 2. These all use multiple monitors and halting filtering. The various communication schemes are implemented by calling different dispatch methods in the instrumentation. For per-thread partitioning, a dispatch method is used which automatically tags messages with the identifier of the thread calling the method and for arbitrary partitioning a dispatch method that requires a tag to be given is used.

Table 2 refers to the additional threads used in each architecture, note that m (the number of threads in the monitored system) is determined at runtime and n is a parameter to the architecture specifying how many monitors, running in parallel, the different subtraces, defined by tags, are to be spread between.

4 Experiments

Two sets of experiments were conducted. We first evaluate the Clustered and Tagged architectures against artificial workloads characterising behaviour we would expect to see at runtime (Section 4.1) and then use a set of benchmarks taken from the DaCapo benchmark suite (Section 4.2). Benchmarks were instrumented using AspectJ to dispatch events using MAIL.

The machine used is an Apple Mac Pro with two 2.26GHz quad-core Intel Xeon processors and 16GB of memory. The processors use Intel's hyperthreading technology making 16 hardware threads available, however only 8 hardware threads can be executing at any one time. The JVM options `-server -Xmx16G` are used. Here this means that some elements of garbage collection are concurrent. Java threads are mapped one-to-one onto native threads by the JVM, which are scheduled by the operating system. Note: In the graphs below, as the inverse of running time is used higher points represent shorter running times.

4.1 Microbenchmarks

In this section we evaluate the Tagged and Clustered architectures against characteristic workloads.

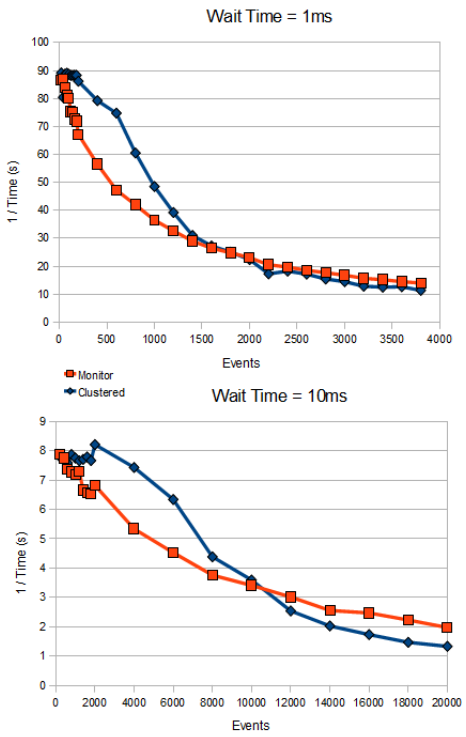

Fig. 3. Clustered Results.

Clustered Architecture - This architecture was developed to utilise the time between synchronisation points, given by fail-possible events, to process events asynchronously. The two factors that effect the performance of this architecture are the number of events in a cluster and the time between the last event in the cluster and the end of the cluster.

The associated microbenchmark is based on the well-known `Iterator` methods protocol stating that after an `Iterator` is created each `next` call should be preceded by a `hasNext` call. A workload is created that creates and uses multiple iterators with an artificial *wait time* between each iterator. The `Iterator` creation event is selected as fail-possible so that when the next iterator is used we know that the previous iterator was used safely. Iterator objects are generally used *intensively* and *infrequently*, which relates to the idea of large widely spread clusters.

Graphs for 1 and 10 millisecond *wait times* are given in Figure 3. As expected, this shows that there is an upper limit on the number of events that a certain amount of time between clusters can cover. The maximum gains are a 1.58 times increase in the 1ms case, and 1.4 times in the 10ms case. Note that only a single additional thread is used, although this approach would not scale with more threads. Reducing the amount of time it takes to monitor each event would increase the number of events coverable by a wait time.

Tagged Architecture - This architecture was developed to expose inherent parallelism in the trace itself. Its associated microbenchmark is designed to investigate how the relationship between the number of tags and events effects performance when there are a large number of tags, a larger number of events and tagged events are heavily interleaved. This heavy interleaving is important as it means that parallel monitors have sufficient work.

This workload constructs a random list of tagged events and then dispatches them via MAIL. The key reason why overhead should be reduced is that this architecture communicates asynchronously and the time between events is maximised, increasing the likelihood that the previous event has been evaluated when the next event is generated.

Graphs for different numbers of events and tags are given in Figure 4. The number of threads indicates the number of monitors running in additional threads. By Amdahl's Law [2], the monitoring work can be decomposed into sequential

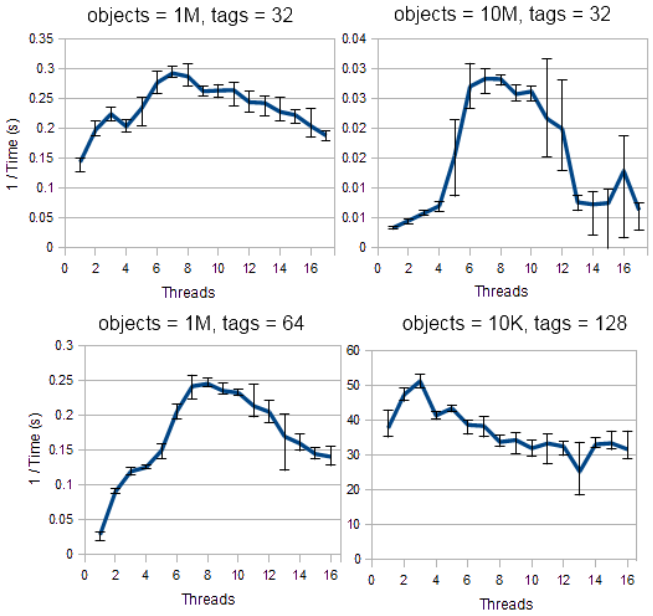

Fig. 4. Tagged Microbenchmark Results.

and parallel parts and we would expect only the parallel element to be reduced, placing a sub-linear limit on speedup. However, the best speedup observed in the presented experiments is 8.38 times with 7 threads (1M events and 64 tags). This apparent super-linear speedup is probably due to the smaller state of each monitor reducing the average time to process a single event. The degradation in performance with additional threads is due to the parallel benefits being overshadowed by scheduling and interference overheads. As the state of all monitors fit comfortably into cache the additional 8 threads given by hyperthreading are not utilised effectively, explaining the drop off after eight threads in these graphs. The large variations are due to the random method for assigning tags.

A correlation can be observed - when there are more tags, the limit on scalability goes up as work can be spread more evenly and when each tag has more work, the magnitude of this speedup increases as the proportion of parallelisable work is greater.

4.2 Benchmarks

We use a selection of benchmarks (Table 3) taken from the 2009 DaCapo benchmark suite [8] and specifications (Table 4) taken from [9]. The suite provides different sized workloads for each benchmark and we selected the largest workload for each benchmark. Table 5 gives the number of events from monitoring the specifications against the benchmarks.

Table 6 gives the results of monitoring the benchmarks for the specifications. Timings come from an average of at least three runs and the `-converge` option is used when running the DaCapo benchmarks, ensuring that benchmark running times converge. The Tagged architecture was evaluated with 1, 2, 4, 8 and 16 monitors only due to time constraints. A base unmonitored running time for each benchmark is given and the results are reported as slowdown from this base time. Monitor indicates the original monitor and runs achieving better than this (a lower slowdown) have been highlighted in bold italic. A dash indicates that the experiment failed to complete within a reasonable time. Therefore, the time

Name	Description	Threads
batik	Produces SVG images based on unit tests in Apache Batik.	2
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.	1
luindex	Uses lucene to index a corpus of data comprising the works of Shakespeare and the King James Bible.	16
lusearch	Uses lucene to carry out a text search on a corpus of data comprising the works of Shakespeare and the King James Bible.	16
pmd	Analyzes a set of Java classes for a range of source code problems.	16
sunflow	Renders a set of images using ray tracing.	16
tomcat	Queries a Tomcat server and verifies the retrieved webpages.	16
xalan	Transforms XML documents into HTML.	1

Table 3. Selected Benchmarks. The Threads column gives the number of software threads the benchmark uses.

Name	Description
FailSafeIter	An iterator created (`create(c,i)`) from a collection should not have `next(i)` called on it after the collection has been updated (`update(c)`).
HashMap	Whilst an object is in a hash map (`add(o,h)`) its hashcode should remain the same for calls to `contains(o,h)` and `remove(o,h)`.
HasNext	Every `next(i)` call to an iterator should be preceded by a `hasNext(i)` call. No violation can occur after `hasNext` is false, given by the `end(i)` event.

Table 4. Selected Specifications.

			Benchmarks							
			batik	fop	luindex	lusearch	pmd	sunflow	tomcat	xalan
Specifications	FailSafeIter	create	31082	7672	64	256	594121	0	27272	0
		next	16352	458413	151	512	3327420	0	30584	0
		update	90732	235500	4191	745034	3733709	0	16066	0
		total	138166	701585	4406	745802	7655250	0	73922	0
	HashMap	add	478	141	27	1408	0	0	72282	258072
		contains	14	0	0	0	0	0	0	0
		remove	0	0	0	0	0	0	0	0
		total	492	141	27	1408	0	0	72282	258072
	HasNext	hasNext	47320	764545	214	768	3885892	3952568	57858	0
		next	16352	458413	151	512	3327420	3666465	30585	0
		end	30968	72624	63	256	558378	285751	27273	0
		total	94640	1295582	428	1536	7771690	7904784	115716	0

Table 5. Number of events for benchmarks monitored against different specifications.

taken to monitor Batik for HasNext using Tagged with 8 monitors is the base time, 4363 milliseconds, multiplied by the slowdown multiplier, 1.68, giving 7330 milliseconds. The results from monitoring each specification are discussed below.

FailSafeIter - It appears that the Tagged architecture has achieved good speedups for this specification, however, a map from collections to iterators was used to generate tags in the instrumentation and this filtered out irrelevant events. This highlights the importance of careful and clever instrumentation to generate only the required events. In places the number of events was significantly reduced - for example all events were removed in the Tomcat benchmark. Consequently, the Tagged architecture achieved good results - with 8 threads Fop was monitored 5.4 times faster. Table 7 gives the spread of tags across monitors for Fop, showing a large load imbalance. The two causes of this are the simple hash function used to assign tags, and the fact that the number of tags is dependent on the number of different collections used to create iterators.

The Clustered architecture synchronises on each `next` event, as this is the event that can cause the specification to fail. This results in a smaller slowdown

		Base Time (millisecs)	Slowdown (multiplier)								
			Monitor	Clustered	Tagged					Async	Post
					1	2	4	8	16		
FailSafeIter	Batik	4363	8.75	15.29	26.03	25.83	46.37	44.93	46.71	14.09	12.88
	Fop	469	127.81	284.17	135.81	***65.05***	***29.8***	***23.42***	***26.68***	397.62	438.99
	Luindex	920	1.41	1.56	***1.38***	1.43	1.45	1.71	2.12	1.53	2.16
	Lusearch	2753	11.9	***3.49***	***7.08***	***10.18***	***7.57***	***7.4***	***7.65***	14.44	14.44
	Pmd	2734	648.12	***361.93***	-	-	-	-	-	-	-
	Sunflow	4908	11.16	40.41	***1.59***	***1.48***	***1.49***	***1.73***	***1.78***	156.42	***1.04***
	Tomcat	4492	9.12	***4.1***	-	-	-	-	-	12.86	13.79
	Xalan	11041	24.42	222.63	***0.83***	***1.17***	***1.54***	***1.94***	***2.57***	232.59	***0.76***
HashMap	Batik	4363	1.47	1.52	1.56	1.59	1.76	2.01	2.46	2.46	2.37
	Fop	469	2.01	2.13	***1.99***	***1.99***	2.02	2.87	2.89	2.1	***1.56***
	Luindex	920	1.32	1.48	1.36	***1.31***	1.42	1.65	2.1	1.5	1.44
	Lusearch	2753	2.15	***1.89***	2.28	***1.63***	***1.68***	***1.88***	2.67	***1.88***	***1.9***
	Pmd	2734	2.56	***1.89***	4.46	4.57	4.67	5.81	6.87	***1.17***	***1.12***
	Sunflow	4908	2.08	***1.1***	***1.36***	***1.63***	***1.49***	***1.65***	***1.63***	***1***	***1.03***
	Tomcat	4492	14.64	***2.24***	***14.11***	***7.74***	***6.11***	***4.18***	***4.26***	***2.18***	***2.18***
	Xalan	11041	100.22	1723.82	***75.77***	***62.28***	***66.71***	104.52	***93.5***	1879.5	-
HasNext	Batik	4363	1.72	2.9	***1.47***	***1.51***	***1.54***	***1.68***	1.95	2.59	2.75
	Fop	469	79.32	179.64	570.6	306.76	211.6	176.14	107.01	474.49	472.52
	Luindex	920	2.94	***1.49***	***2.92***	***1.54***	***2.01***	***2.15***	***2.83***	***1.41***	***1.91***
	Lusearch	2753	1.38	***0.82***	***0.73***	***0.77***	***0.77***	***0.8***	***1.28***	1.6	1.96
	Pmd	2734	32.62	77.47	399.71	180.71	95.36	65.96	44.54	275.26	437.39
	Sunflow	4908	9.11	34.83	316.21	158.53	108.97	72.24	17.79	261.81	246.94
	Tomcat	4492	2.34	3.76	2.93	2.92	2.93	3.59	4.21	9.66	7.22
	Xalan	11041	1.65	***1***	1.89	2.01	2.08	2.11	2.13	***1.01***	***1***

Table 6. Experimental results for the MAIL architectures.

for benchmarks where the iterator to usage event ratio is low - i.e. there are few events in a cluster. This supports the findings in Section 4.1 that the magnitude of speedup is limited by this. Tomcat saw a 2.2 times speedup and Lusearch a 3.4 times speedup.

Lastly, as the Sunflow and Xalan benchmarks do not create any events for this specification, it would be expected that their monitoring with different architectures would create a consistently small amount of slowdown. However, the slowdown observed varied greatly. The large overheads are caused by work carried out in the instrumentation to decide whether to create an event. The variation in overhead is dependent on the interfering threads created even when no events are monitored. Consequently, the Post architecture achieved 10.7 and 32 times speedup for Sunflow and Xalan respectively.

	Fop-FailSafeIter															
1	100															
2	32.25								67.65							
4	16.29				52.07				15.94				15.7			
8	8		43.99		8.09		7.78		8.2		7.88		7.95		8.12	
16	4.04	39.74	3.8	3.88	3.95	4.05	3.73	4.04	3.91	4.28	4.13	3.96	4.4	3.77	4.27	4.04

	Xalan-HashMap															
1	100															
2	59.99								40.01							
4	59.99				0				0				40.01			
8	19.99		0		0		26.67		40.01		0		0		13.34	
16	13.32	0	0	13.33	13.33	0	0	0	6.67	0	0	13.33	26.67	0	0	13.33

Table 7. The average spread of tags across 1,2,4,8 and 16 monitors, as a percentage

HashMap - Because of the relatively few events for this specification, the slowdowns are generally not large, with the obvious exceptions of Tomcat and Xalan. Tomcat worked well with Async and Post architectures, achieving speedups of 6.7 times and with 2 monitors the Tagged architecture achieved a speedup of 1.5 for Xalan. Table 7 shows that there is substantial load imbalance when using the Tagged architecture for the Xalan benchmark which explains why this architecture did not perform as well as might be expected considering the large number of events. Where slowdown is below zero the benchmark ran faster when monitored. This effect is most likely caused by additional instrumentation and monitoring code causing the JIT compiler to perform different optimisations, and threads to interleave in different ways. This also occurs with detrimental effect elsewhere, and the overhead is not entirely from the monitoring but also from interference with the monitored application - this was the reason for developing the Post architecture.

HasNext - Figure 5 indicates the scalability of the Tagged architecture for selected benchmarks is given in. When comparing this to the number of events reported in Table 5 we can see that for benchmarks with large numbers of events (Pmd, Fop and Sunflow) the architecture scales better than benchmarks with fewer events (Batik, Luindex and Lusearch) - the gains in evaluating these events concurrently make up for the extra costs involved in running additional

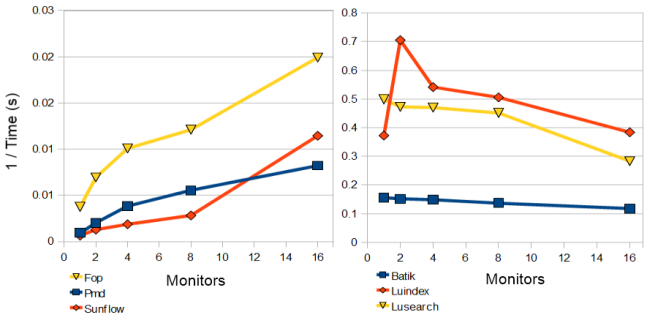

Fig. 5. HasNext Tagged Results.

monitors. Fop was monitored 5.3 times faster with 16 monitors than with one. Whilst the benchmarks with a large number of events scaled well, the overall slowdown was much larger than when using the monitor by itself. MAIL adds an additional overhead per event and this is greatest in the Tagged architecture, as every event must be tagged and the tag then used to lookup the relevant monitor. This overhead could be reduced by removing the layer of indirection in MAIL, among other possible optimisations. It is not obvious that this overhead is reducible to the extent that this architecture will consistently perform better than others - this is future work.

Using the end event, it is possible to make an estimation as to the average number of iterations carried out per iterator, as reported in Table 8. This indicates why the Clustered architecture may not have been very effective, as there is only a small number of events per synchronisation point.

	batik	fop	luindex	lusearch	pmd	sunflow	tomcat
Iterators	30968	72624	63	256	558378	285751	27273
Average Iterations per Iterator	0.53	6.31	2.4	2	5.96	12.83	1.12

Table 8. Average iterations per iterator for HasNext specification.

Summary. No one architecture performed consistently better than other architectures. The Tagged architecture scaled well with a large number of events and it was noted that load balancing is a problem. The Asynchronous architecture was generally less efficient than the Clustered architecture. This seems unintuitive at first but demonstrates the effect that not clearing garbage rule instances can have on the event processing time. The Post architecture showed that in some cases delaying monitoring can reduce overhead, by reducing interference.

5 Conclusion

In this study, we describe a communication layer, MAIL, which provides a general-purpose platform to explore approaches to parallelising the runtime monitoring approach, specifically the interaction between monitor and application. We have shown:

- Significant performance improvements for the two main approaches (Clustered and Tagged) for characteristic workloads,
- Mixed results for the DaCapo benchmark suite across all approaches,
- That MAIL is an appropriate general interface for conducting experiments into how the interaction between monitor and application can be parallelised.

Overall, we conclude that multicore systems can reduce the overhead of runtime monitoring. Importantly, we note that the level of reduction is dependent on the level of data-parallelism in the trace, which is itself dependent on a combination of the monitored application and specification.

There are two reasons why the results for the DaCapo benchmarks were less convincing. Firstly, the benchmarks consist of Java applications all displaying similar workloads for the given specifications. These workloads do not exhibit the level of data-parallelism we would expect when monitoring other systems -for example, when checking whether a web server processes a request correctly would exhibit many interleaved tags with clusters of activity. Secondly, due to the generality of MAIL the implementation has not been made as efficient as possible, for example to allow different monitors to be used a level of indirection is introduced and the same mechanism is used for different tagging schemes.

This investigation used RULER monitors but other monitors could be used, raising the question of how dependent these results are on the monitor used. We note that for the implementation of RULER monitors used here the average processing time is heavily dependent on the number of live rule instances in the monitor. This makes average event processing times dependent on the trace and garbage removal techniques in ways other monitors may not be. Using a different monitor would not only see a scaling of average event processing time, but other factors such as interference and garbage collection would play a role. As a consequence, it is difficult to say how using other monitors would be reflected in speedups achieved and this is an important piece of further work.

This is an initial investigation and it is clear that more work is possible. We consider the following as the main areas for future investigation :

1. *To what extend does the potential for parallelism depend on the runtime monitoring system* - What is the relationship between the implementation of the monitor and the speedups achievable through parallelism.
2. *What determines the level of parallelism* - As the level of data-parallelism in the trace (spread of synchronisation points and interleaving of tags) dictates the benefits of parallelism we should investigate what determines this.
3. *Automatically finding parallelism* - To make this approach more applicable we should explore if fail-possible events and tagging schemes could be inferred from the specification itself and different parameters to these methods adapted dynamically.
4. *Partitioning the trace in time* - When there are many events between synchronisation points (For example in Log-File Analysis), the trace could be partitioned in 'time', or across the length of the trace. The applicability of this approach would depend on appropriate decompositions of specifications in the monitoring logic.
5. *Tackling interference* - The monitor may interfere with the application through shared resources, this is potentially increased by parallel monitors running on the same machine. By separating the monitor and application onto different (virtual) machines, increased latency could be traded for reduced interference.

References

1. 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. *SIGPLAN Not.*, 40:345–364, October 2005.
2. G. M. . Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. *Proc. Am. Federation of Information Processing Societies Conf.*, AFIPS Press:438–485, 1967.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
4. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 2010.
5. H. Barringer and K. Havelund. A Scala DSL for trace analysis. *17th International Symposium on Formal Methods (to appear)*, 2011.
6. H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. *Runtime Verification, 9th International Workshop, RV 2009*, pages 1–24, 2009.
7. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RuleR. *J Logic Computation*, 20(3):675–706, June 2010.
8. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, ANU, 2006. http://www.dacapobench.org.
9. E. Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009. Available in print through ProQuest.
10. F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA '07*, pages 569–588, New York, NY, USA, 2007. ACM.
11. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS '09*, pages 246–261, Berlin, Heidelberg, 2009. Springer-Verlag.
12. C. Colombo, G. J. Pace, and P. Abela. Compensation-aware runtime monitoring. In *RV*, pages 214–228, 2010.
13. N. Delgado, A. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 2004.
14. D. Drusinsky. The temporal rover and the ATG rover. In *Proceedings of the 7th International SPIN Workshop*, pages 323–330, London, UK, 2000. Springer-Verlag.
15. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 135–, Washington, DC, USA, 2001. IEEE Computer Society.
16. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *PDPTA*, pages 279–287, 1999.
17. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
18. P. Meredith and G. Roşu. Runtime verification with the RV system. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 136–152, Berlin, Heidelberg, 2010. Springer-Verlag.
19. G. Reger. Rule-based runtime verification in a multicore system setting. Master's thesis, School of Computer Science, University of Manchester, 2010.
20. V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electron. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006.