Operating System Level Trace Analysis for Automated Problem Identification

Gabriel N. Matni and Michel R. Dagenais
Department of Computer and Software Engineering
Ecole Polytechnique de Montreal
C.P. 6079, Station Downtown, Montreal, Quebec
Canada, H3C 3A7
Tel. +1 514 340 4711/4029
Fax +1 514 340 3240
{gabriel.matni, michel.dagenais}@polymtl.ca

ABSTRACT

Performance bottlenecks, malicious activities, programming bugs and other kinds of problematic behavior could be accurately detected on production systems if the relevant events were being monitored. This could be achieved through kernel level tracing where every time a relevant event occurs, the information is analysed or saved in a trace file to be inspected during post-mortem analysis. While collecting the information from the kernel has a very low impact, the offline analysis is typically performed remotely with no overhead on the system whatsoever.

This article presents an automata-based approach for analyzing traces generated by the kernel of an operating system. Some typical patterns of problematic behavior are identified and described using the State Machine Language. These patterns are fed into an offline analyzer which efficiently and simultaneously checks for their occurrences even in traces of several gigabytes. The analyzer achieves a linear performance with respect to the trace size. The remaining factors impacting its performance are also discussed. The main interest of the proposed approach is the efficiency obtained in monitoring such extensive and detailed execution traces for a very large number of simultaneous possible patterns of problematic behavior.

General Terms

Performance, Security, Languages, Verification

Keywords

Finite State Machines, Kernel Tracing, Trace Analysis, Pattern Matching, Performance Debugging

1. INTRODUCTION

By carefully examining execution traces of a computer system, experts can detect problematic behavior caused by software design defects, inefficiencies as well as malicious activities. Kernel tracing can often reveal the main source of such problems. Tracing consists in instrumenting the kernel code to precisely record its behavior at execution time. Typical kernel events traced include all system calls from processes, scheduling events, interrupts, I/O operations and may include locking operations.

It is now possible to achieve low overhead, low disturbance tracing of multi-core Linux systems with the Linux Trace Toolkit next generation (LTTng). It provides precise, low impact, highly reentrant tracing and is used for efficiently debugging large clusters [7] as well as narrowing time constraints problems in real-time embedded applications [13]. The information about the filesystem, inter-process communication, system calls, memory management and networking is efficiently collected, precisely time stamped and saved at runtime. This information is used to debug the monitored system and a large class of problems may be detected, such as excessive disk swapping, excessive threads migration, frequent writes of small data chunks to disk, locking problems, security problems and many others. Once the execution trace is available, the objective is thus to automatically validate it against a pool of predefined problematic patterns.

The novelty of the approach lies in the application of powerful patterns to detailed operating system level traces. The applications run unmodified, with a minimal overhead imposed by the operating system level tracing. A particular emphasis was placed on performance, given the detailed level of the traces (their size potentially in tens of gigabytes), the increasing number of parallel cores in systems, and the multiple patterns to be checked simultaneously.

Motivations and Goals. Execution traces can either be analysed on the fly in memory or offline (on a different system or a posteriori on the same system). On the fly analysis may obviate the need to store on disk the execution trace and may be interesting from an overhead point of view if a small number of simplem patterns are searched for; in that case the pattern searching may be faster than storing the trace. For many applications, however, it is interesting

to store the trace anyway. The stored trace may indeed be used to dig into an issue further when a problem is detected. Moreover, in many cases, it may be desirable to search for a large number of complex patterns without impacting much the performance of the studied system. For these reasons, an offline system was implemented. It should be noted, however, that the algorithms presented in this article work in a single pass and would thus be just as well applicable to on the fly analysis.

The most popular kernel trace analysis tools that help simplify the debugging task provide offline event filtering and trace visualization. These tools include LTTV [17], QNX Momentics [2] and Windriver Workbench [4]. Offline filters are used to highlight the events of interest satisfying a set of constraints. Visualizers, such as the Gantt chart of the control flow view (e.g. LTTV [17]), help the developer seek throughout the trace and determine visually any sort of unexpected behavior. Even when these tools are used, validating the existence of a set of problematic patterns in one or several large traces remains a manual and time consuming task. This motivated the development of an automated approach to represent patterns of problematic behavior and to automatically and simultaneously check for their existence in one or several large traces.

1.1 Related Work

Frequent pattern mining for kernel trace data [16] is a recent work aiming at the detection of recurring runtime execution patterns, such as inter-process communication patterns. The work finds the set of all temporally proximal events that occurred frequently in a trace. This helped identify the processes that are heavy consumers of system resources but still remain invisible to traditional tools such as top. This approach is interesting but doesn't allow validating the trace against a set of predefined patterns which may occur very rarely in the trace.

Systemtap [15] and DTrace [8] provide scripting languages resembling C that are used to enable probe points in the kernel (instrumentation sites) and to implement their associated handlers. These handlers could be used to perform run-time checking and to generate warnings when something bad happens. The script file is translated into C code and then compiled into a binary kernel module. These C-like scripting languages do not provide a simple way to describe complex patterns at a high level of abstraction. While this approach implements the basic instrumentation mechanism, it does not provide a framework to either perform pattern searching or to store the event data in an execution trace for later analysis.

In parallel computing, many tools exist that are able to automatically detect performance problems in MPI, OpenMP or hybrid applications. These tools include Paradyn [18] and EXPERT [21]. EXPERT instruments the application's source code so that a trace file in the EPILOG format is generated upon running the program. The performance patterns are supplied to the tool and are written as Python classes implementing a common interface, making them exchangeable from the perspective of the tool. These pattern classes register callback functions for every event of interest and are capable of accessing additional events by retrieving the updated state information or by following some event dependencies. In the system that we propose, the patterns coded using the State Machine Language can similarly access the updated system state maintained by the LTTng Viewer.

Using Finite State Machines to describe patterns is found in the field of network based Intrusion Detection, particularly in misuse detection systems or scenario-based systems. The State Transition Analysis Technique (STAT) [14], developed at University of Santa Barbara, is used to model computer penetrations with Finite State Machines (FSM) patterns called attack scenarios. Each scenario is composed of states and transitions. Transitions are triggered by the occurrence of particular events on the network and can take the system from an initial safe state to a final compromised state. The main features of the STAT language such as transition guards and actions are also found in the State Machine Language [3] which we will be using in our work because of its open-source implementation. By the time the article [14] was written, around 35 attack scenarios were described using the STAT language, and the authors claim that no limit in the expressiveness of the language was found. Furthermore, recent work has been done to automatically translate the large collection of rules written for SNORT - a popular intrusion detection system - into STATL scenarios. We believe the STAT approach could be very well applicable to kernel traces for many reasons. First the objective is quite similar; in both cases, patterns are composed of a sequence of events that could be used to describe either security threats or performance problems. Secondly, the automata-based approach provides an easy way to describe complex patterns from multiple simple ones, through the creation of synthetic events.

Ragel [19] is a popular state machine compiler used mainly to generate lexical analyzers and to validate user input. The generated code tries to match patterns to the input, favoring longer patterns over shorter ones. The Ragel language provides four types of transition actions. They allow the FSM developer to execute a particular action whenever the state machine transitions from one state to another. However, none of the provided actions allow us to explicitly assign different actions for different transitions, from any state in the FSM.

2. FAULTY BEHAVIOR

While the system will be easily extensible at a later time, it was important to start by collecting a representative set of problematic patterns touching on several fields such as security, software testing and performance debugging. For sake of brevity, a representative subset is described here.

2.1 Security

The SYN flood attack is a denial of service attack that consists in flooding a server with half-open TCP connections. Signs of a SYN flood attack may be found in a kernel trace if the relevant events were instrumented. It would be very inefficient to manually look for patterns caused by such an attack, thus the interest in automating the lookup process.

Escaping the chroot jail is another attack type that can be caught on a system: a privileged process (euid=0) may want

to confine its access to a subtree of the filesystem by calling the chroot() system call, immediately followed by the call chdir("/") to setup the chroot jail. If a process ever tries to open a file after the call to chroot(), without a chdir("/"), then this is considered to be a security vulnerability [9]. Indeed, a malicious user can trick the program to open the system file ../../../../../etc/shadow, for example.

Even though they are rare, Linux viruses do exist and they could be detected on a traced system. The approach we propose is different from the ones used in anti-virus software. An instrumented Linux kernel records all interactions with the operating system. This interaction, consisting in a sequence of system calls, includes the behavior of possible viruses, to search for when analyzing traces generated from production systems. For example, in [12], the virus Linux.RST.B was observed generating the following actions: it executes a temporary file ".para.tmp" which creates three other processes; it opens and lists the current directory and modifies the binary files in /bin. By analyzing a kernel trace, it should be possible to detect a viral behavior automatically, while diagnosing at the same time some other security and performance problems.

2.2 Software Testing

Shared resources often require locks to be held before accessing them, to avoid race conditions. In the Linux kernel, locking is more complex than in user-space, due to the different states the kernel could be in (preemption enabled, disabled, servicing an irq, etc.). Validating each and every lock acquire has already been implemented in lockdep, the Linux kernel lock validator [1]. For instance, it makes sure at run-time that any spinlock being acquired when interrupts are enabled has never been acquired previously in an interrupt handler. The reason is that the interrupt could happen at any time, in particular when the spinlock is already held, deadlocking therefore the corresponding CPU. Activating this option requires recompiling the kernel and adds a continuous overhead on the system. Instead, using a kernel trace and a posteriori analysis, the same kind of validations may be performed.

Another detectable programming bug consists in accessing a file descriptor after it was closed. This illustrates a more general class of programming errors where the usage specifications state that two particular events are logically and temporally connected.

2.3 Performance Debugging

Some inefficiencies in software could be detected from I/O events. For instance, frequent writes of small data chunks to disk would impact the overall system performance and are to be avoided. Similarly, reading the data that was just written to disk, or reading twice the same data, or even overwriting the data that was just written, are all signs of inefficiencies that are visible in a kernel trace.

Multimedia applications, and more generally soft real-time applications, are characterized by implicit temporal constraints that must be met to provide the desired QoS [5]. Assuming that tracing the kernel scheduler has a negligible impact on the system, we can verify that temporal constraints are satisfied for one or multiple real-time applications, and whenever they are not, we can show what the system was doing at that time.

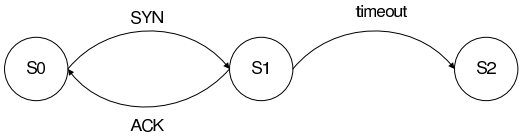

Figure 1: Detecting half-open TCP connections

3. AUTOMATA-BASED APPROACH

We first describe in 3.1 the state machine language and we show how it was used to model the three following scenarios: chroot jail escape, locking validation and real-time constraints checking.

3.1 SM Language

Describing the various patterns using the SM Language [3] is straightforward. Even though many existing languages are capable of expressing the different scenarios described in section 2, a state-transition language was selected for the following reasons:

1. **Simplicity and expressiveness:** the language is easy to use and provides enough features to express new, yet to be defined, scenarios [14].
2. **Domain independent:** the language may be tailored to support a wide range of patterns that relate to different fields. In the Intrusion Detection field, state-transition language is widely used to model attack signatures [20], [14]. In model checking and Software Security, it is equally used for scenario-oriented modeling to examine security properties [10], [11] or to verify and validate software use cases [6].
3. **Synthetic events:** the state-transition approach lets us easily generate synthetic events from lower level primary events [14]. Consider for instance the SYN flood attack detection. We first model a half-open TCP connection using the state machine shown in Figure 1. When the server receives a connection request, the system moves to state S1. The server sends the acknowledgment and a timer is started. If the client sends back the acknowledgment, the system returns to state S0. Otherwise, when the timeout occurs, the system moves to S2 and a synthetic event is generated called "halfopentcp". Frequent occurrences of this synthetic event would probably mean that an attack is taking place. Synthetic events are very useful when describing even more complex patterns.

The State Machine Language supports the declaration of a state and the transitions originating from it. Each transition has a name, an optional argument list, an optional transition guard, a destination state and a transition action. The guard is a boolean expression written in the target language source code and copied verbatim into the generated output. As such, the guard can do much more than simply associate the transition with an event type, it can contain arbitrarily

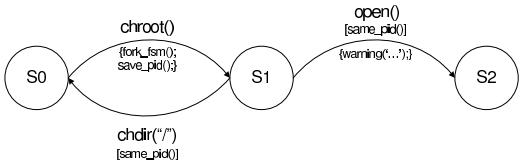

Figure 2: Escaping the chroot jail

```
S1{
chdir(pid: int, newdir: char *)
      [same_pid(pid) && check_new_dir(newdir)]
      S0
open(pid: int)
      [same_pid(pid)]
      S2
      {warning(pid); destroy_fsm();}
}
```

Table 1: SM Code Snippet

complex logic like testing properties of the event or of the system state. If the expression is evaluated to true, then the transition is triggered and the transition action is executed. The destination state could be defined in another state machine declared in another file for simplicity. The transition actions are functions implemented in the target language and could have a regular argument list. Similarly, every state can have on-entry actions as well as on-exit actions than could be useful to start/stop a timer or update some internal data structures.

3.2 Escaping a chroot jail

An automaton showing the sequence of system calls that may result in a security violation is shown in Figure 2. The vulnerability is explained in 2.1. A call to chroot() brings the system to state S1 and saves the process id. Furthermore, a new FSM is forked in case a new chroot() call is issued by another process. The FSM fork is initiated by the transition action fork_fsm(). Any process issuing a successive call to chdir("/"), brings back the corresponding FSM to state S0, whereas a call to open() brings it to S2 and generates a warning. The machine transitions to a fourth Exit state, not shown here, and it happens whenever the exit() call is issued by the process.

We show in table 1 a self explanatory code snippet of the language describing state S1 from Figure 2. From state S1, two transitions are possible, chdir() and open(). If the encountered event is a call to chdir, then the transition guard (between square brackets) is evaluated. In this case, if the functions same_pid() and check_new_dir() return true, then the transition is triggered and the system moves back to state S0. It is also possible to have a transition action (between braces). In our example, the call to the function warning() occurs only if the corresponding transition guard is evaluated to true.

3.3 Locking Validation

We generate in Figure 3 an automaton that will validate a subset of the kernel locking rules. The event irq_entry() brings the system to state Irq_Handling and event irq_exit() brings it back to its normal state. Any lock could either

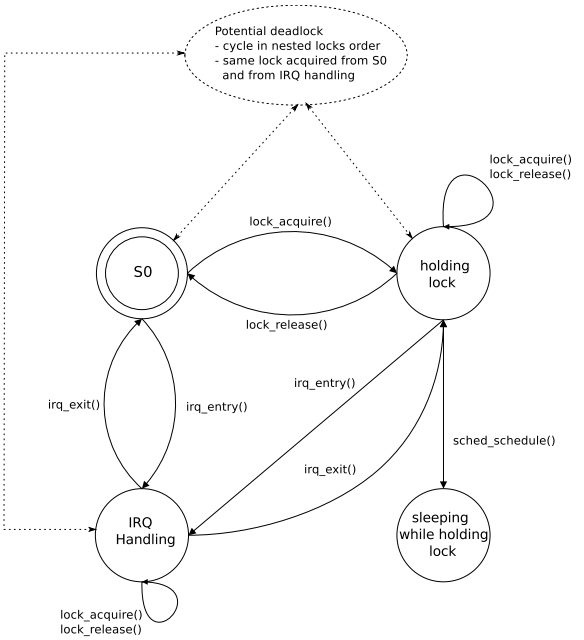

Figure 3: Locking Validation

be acquired from the normal state (S0 or Holding_Lock) or the Irq_Handling state. If a lock being acquired when interrupts are enabled has been previously acquired from the Irq_Handling state, the system transitions to state Potential_Deadlock. The reason is that once this lock is taken and before it gets released, if the code is interrupted by the same handler which tries to acquire the same lock, then a deadlock occurs. Similarly, if a lock previously taken when irqs were on, is now being acquired from an irq handler, then the system should also transition to the state Potential_Deadlock.

Suppose the system is in state Holding_Lock on a particular processor, where a lock is being held on behalf of a certain process. If this process gets scheduled out, then there is another potential deadlock due to the fact that some other process may require the same lock.

Nested locks, taken on behalf of the same process could deadlock the system if they are not taken in the right order. When the system is in the state Holding_Lock, the arrival of a new event lock_acquire would trigger the corresponding transition. This results in a call to a function that generates trees of lock dependencies implemented in a hashing table. At the end of the analysis, if a cycle is found, then there is a potential deadlock and the involved locks are shown. The return address, which is a traced event argument, can help identify the code section responsible for holding the lock.

During our experiments, an interesting case was found in function copy_pte_range(), in mm/memory.c in the Linux kernel, which generated a cycle in our analysis. The suspi-

```
static int copy_pte_range(struct mm_struct *dst_mm,
  struct mm_struct *src_mm, ...)
{
     ...
     spinlock_t *src_ptl, *dst_ptl;
     ...
     spin_lock(dst_ptl);
     ...
     spin_lock_nested(src_ptl, ...);
     ...
}
```

Table 2: Suspicious Code Sequence

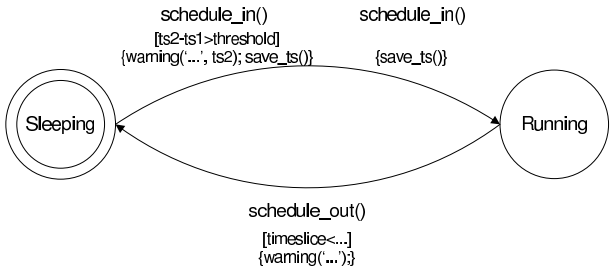

Figure 4: Real-Time Constraints Checking

cious code sequence that caused the problem is abstracted in Table 2. The function receives pointers to two mm_struct structures and always locks the destination page_table_lock spinlock, followed by the source lock. If another CPU is doing the copy but with the reversed parameters, then the locks would be taken in the opposite order and a deadlock can occur. After further investigation, we noticed that a call to this function is initiated by a call to copy_process() in fork.c which is called when forking a process. This function calls dup_mm() which allocates memory for a new mm_struct becoming the dst_mm shown in Table 2. Since no other processor could be using the newly initialized structure as being the src_mm in function copy_pte_range(), there is no potential deadlock. However, this shows how our approach was useful to identify suspicious code sequences.

3.4 Real-time Constraints Checking

To support soft real-time applications, the kernel should respect the application's temporal constraints and therefore a predictable schedule is desired [5]. Such applications may require periodic scheduling where the period is derived from the frame rate of an audio/video stream, for example. We show in Figure 4 a detailed state machine that enables us to check if the application's execution period has been respected throughout the life of the trace. Whenever it's not, we show the list of events that hindered the application's scheduling.

From state Sleeping, the transition schedule_in() brings the FSM to the Running state and saves the event time stamp; it also computes the difference between every two consecutive schedule_in() events. If the result is greater than a user specified threshold, a warning is generated. The event time stamp displayed by the warning() call, can then be used to reach and scrutinize the preceding events once the trace is opened using the Linux Trace Toolkit Viewer (LTTV). From the Running state, the event schedule_out() brings the FSM back to the Sleeping state. The time stamp of this event is also used to compute the assigned time slice for the application so that the transition could also trigger a warning when the time slice is less than expected.

4. IMPLEMENTATION

We used the Linux Trace Toolkit LTTng, a low-impact, open-source kernel tracer, to instrument the kernel events required by the patterns description. We used the SMC compiler to generate C code for the state machines written in the SM language. The compiler is an open-source java program that supports code generation in 14 different languages.

For every event required by a given pattern, the analyzer registers callback functions with the trace reader and visualizer program LTTV. The program reads the trace sequentially in one pass. When a registered event is encountered, the analyzer calls the corresponding transition for every related state machine. There, if the transition guard is evaluated to true, the transition action is executed before entering the destination state and returning control to the analyzer.

In some cases, when a transition is triggered, a new FSM of the same type needs to be forked. This is referred to as a non-consuming transition type in STATL terminology (see example in 3.2). Whenever required, a transition action can request a fork from the analyzer, generating therefore a new instance of the FSM.

In other cases, such as the locking validation pattern, one finite state machine per CPU is enough. There, the analyzer determines on which CPU the event occurred, and only calls the transition of the FSM for that particular CPU.

The FSM approach offers great flexibility to model, update and optimize one or several patterns. When we instrumented the events of interest for the locking validation pattern, we noticed that the irq entry and exit events are not needed because the information could be determined from the lock_acquire() event. At this point, we simply eliminated the Irq_Handling state from our FSM.

5. PERFORMANCE

The performance of the proposed trace analysis procedure should depend on the number of events in a trace (i.e. trace size) and the number of possible transitions to evaluate in simultaneous active finite state machines (the number of coexisting finite state machines and the frequency of relevant events that may trigger a transition). We instrumented the Linux kernel version 2.6.26 using LTTng and the tests were performed on a Pentium 4 with 512 MB of RAM. For the tests, several paremeters were varied independently in order to evaluate their effect on performance.

In the first test, three different patterns were searched in traces of varying size (500MB, 1GB, 1.5GB, 2GB). Table 3 presents the execution time of our analyzer to look up 3 different patterns: real-time constraints, file descriptors and the chroot patterns. These results show that the execution time is linear with respect to the trace size. The number of coexisting finite state machines depends on the pattern in

	500 MB	1GB	1.5GB	2GB
rt checking	55s	117s	168s	252s
fd_checking	57s	119s	166s	266s
chroot_checking	55s	108s	166s	266s
all	67s	123s	184s	279s

Table 3: Performance Results

question. For instance, checking the file descriptors usage required one FSM per process accessing one file descriptor, whereas the chroot pattern needs one FSM per process, the locking validation pattern needs one FSM per CPU, and the real-time checking requires just one FSM for the Movie Player (mplayer) process.

Interestingly, the execution time for searching each pattern does not vary much and even checking for all three patterns simultaneously is only slightly longer. This is explained by the fact that reading through the whole trace, to find relevant events, already takes a significant amount of time. Then, depending on the patterns searched, additional execution time is required to run relevant events through the simultaneous FSMs.

Thus, if the time to read through the 500MB trace is C and the time to search in FSMs is respectively for patterns real-time, RT, file descriptors, FD, and chroot, CH, we can isolate each component using the results for searching each pattern and then for searching simultaneously for all patterns. Given, from the first column of Table 3, that $C + RT = 55s, C + FD = 57s, C + CH = 55, C + RT + FD + CH = 67$, we can deduce that $C = 50s, RT = 5s, FD = 7s, CH = 5s$.

We could have expected that the file descriptor pattern, requiring one FSM per process accessing one file descriptor, would be significantly more costly than the real-time constraint, requiring a single FSM. However, the execution time is similar due to the fact that event sched_schedule() (relavant for the real-time pattern) was occurring much more frequently than events read() and write() (relevant for the file descriptor pattern).

Table 4 presents the performance of the analyzer when validating the file descriptor pattern against traces of different sizes, and compares it with the analyzer's performance without invoking the FSMs, but only registering empty callback functions for the 6 events of interest. This is useful to isolate the time required to check the patterns from the time needed simply to get the relevant events from the trace. In addition, two interesting metrics are provided with this test, the number of relevant events and the number of simultaneously active FSMs.

The traces used for this test were generated using two different loads. The first 4 traces were generated while running dbench as a server for 1 client. Dbench is a widely used file oriented benchmark. It recreates the file operations required on a typical file server to serve desktop clients. The last trace was generated while the GNU C Compiler, gcc v.4.2.0, was compiling itself. The relevant events for the file descriptor pattern are the following system calls: close(), open(), read(), write() and dup(), as well as the process_exit() kernel event. The slowdown for each test is computed by comparing the execution time between the two configurations of the analyzer (empty callbacks on relevant events versus checking the patterns using the relevant events).

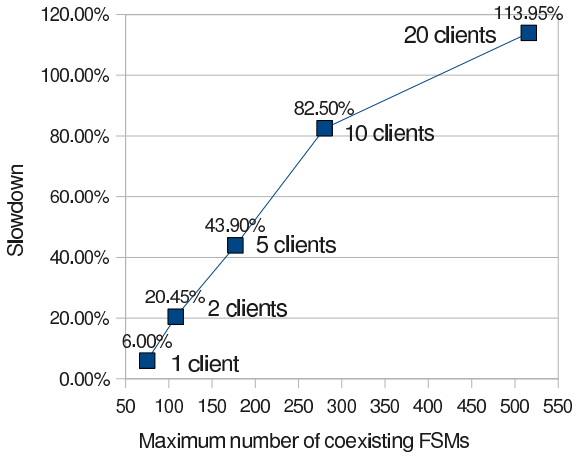

Figure 5: Fixing trace size to 500MB, varying the number of dbench clients

The analysis time for the same trace length differs significantly between the two tests, dbench vs gcc. Indeed, the slowdown was much higher for the gcc trace, even though it contained fewer relevant events than the other traces of similar size. The computed slowdown suggests a direct correlation with the maximum number of coexisting finite state machines handled by the analyzer. For instance, the gcc compilation generated much more (around 50 times) coexisting FSMs than running one dbench client, due to the numerous processes (accessing different file descriptors) generated by the compilation makefile. This resulted in a larger impact on the analyzer's performance.

In the third test, the trace size was fixed to 500 MB and the number of dbench clients varied from 1 to 20. The number of clients is directly proportional to the maximum number of coexisting FSMs during the analysis. The results in Figure 5 show the slowdown percentage with respect to the maximum number of coexisting FSMs in the analyzer, for traces of the same size. The slowdown is directly proportional to the maximum number of FSMs handled by the analyzer. This is expected because the analyzer invokes sequentially all the FSMs in the list for every relevant event, whether the event is needed at the FSM's current state or not.

In a separate test, the performance of the pattern checker, the code generated from the State Machine Language description, was evaluated using the locking validation pattern. The locking validation pattern is indeed most demanding because of its complexity and very high frequency of locking and unlocking events. A hand-written locking validation pattern checker was written in C. It stores the locking state and updates it at each locking and unlocking event. Its algorithmic complexity is expected to be the same as the checker generated from a higher level SML description. However, being hand-written and statically linked as a dedicated application, it avoids some of the indirection caused by the more generic pattern checking machinery.

		relevant events (millions)	coexisting FSMs	Ex. time invoking FSMs	Ex. time empty callbacks	slowdown
1 dbench client	500 MB	2.4	75	51s	50s	6.00%
	1 GB	**4.8**	**72**	**92s**	**86s**	**6.98%**
	1.5 GB	6.9	104	143s	114s	25.44%
	2.3 GB	**11.1**	**72**	**215s**	**189s**	**13.75%**
	3 GB	14.1	104	285s	250s	14.00%
	4.5GB	**18.5**	**83**	**369s**	**338s**	**8.87%**
gcc	2.5 GB	5.7	5241	853	227s	275.77%

Table 4: Slowdown of the analyzer due to FSM invocation with respect to its performance with empty callbacks

Interestingly, the performance of the generated FSM checker was only 4.5% slower than the dedicated hand-written version for the locking validation pattern, a worst-case most demanding pattern. It was expected that a hand-written checker would be faster, especially for the locking validation pattern. However, the small difference is, in our opinion, easily offset by the gain provided by the ability to model patterns at a higher level.

6. CONCLUSION

We presented an automata-based approach to describe some generic patterns of problematic behavior that might occur on production systems. The generated finite state machines can be easily maintained, expanded or even be used as synthetic events to model more complex scenarios. We implemented an analyzer that validates the existence of such patterns simultaneously in large traces and in one pass.

The main contribution of this work was to design, implement and demonstrate a working system capable of obtaining a low overhead detailed execution trace of a production server, and efficiently check the resulting trace for numerous patterns in near real-time. This is possible because of the extremely efficient algorithms used both for the low overhead tracing and for the pattern detection. The proposed architecture and pattern language are efficient and simple to use, and have been demonstrated with a number of real and highly representative patterns.

The analyzer's performance depends greatly on the nature of the patterns being validated. When dealing with a large number of FSM instances of the same pattern, the analysis time is directly proportional to the number of coexisting FSMs and the number of relevant events. By carefully selecting which events to trace, it may be possible to optimise the execution time. For instance, the first version of the locking validation pattern required the events enable_irq() and disable_irq() to deduce in which context a given lock was acquired. It turned out that this information is available at the site where the lock is being acquired. This reduced the number of events to trace, resulting in a smaller trace and a faster analysis.

Another factor impacting the performance of the analyzer is the following; consider the locking validation pattern in Figure 3. Even when the current FSM state is S0, every encountered sched_schedule() event would result in calling the corresponding transition which is irrelevant in state S0. This will call a default transition which maintains the current state and returns control to the analyzer. Instead, the analyzer could have skipped this step since, from the current state, there is no transition sensitive to the event sched_schedule(). This could be achieved by dynamically adjusting the definition of relevant event depending on the current state for a FSM; the analyzer would compute beforehand the list of events leading to state transitions for each state.

The proposed approach is highly parallelizable. It could be used for online near real-time pattern matching of an extensive set of patterns, for monitoring very sensitive servers (e.g. high security applications, extensive test procedures). Further explorations would be useful to support the definition and use of synthetic events. This will allow synthesizing more complex scenarios from multiple simple ones.

7. ACKNOWLEDGEMENTS

The financial support of NSERC is gratefully acknowledged.

8. REFERENCES

[1] The kernel lock validator. http://lwn.net/Articles/185666. Retrieved on 2009-03-10.

[2] Qnx. http://www.qnx.com. Retrieved on 2009-03-12.

[3] The state machine compiler. http://smc.sourceforge.net. Retrieved on 2009-01-22.

[4] Windriver. http://www.windriver.com/products/workbench. Retrieved on 2009-03-12.

[5] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of linux. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.

[6] M. Barnett, W. Girieskamp, and Y. Gurevich. Scenario-oriented modeling in asml and its intstrumentation for testing. In *Foundations of Software Engineering*, 2003.

[7] M. Bligh, M. Desnoyers, and R. Schultz. Linux kernel debugging on google-sized clusters. June 2007.

[8] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, 2004.

[9] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[10] H. Chen and D. Wagner. Mops: an infrastucture for examining security properties of software. 2002.

[11] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[12] M. Desnoyers and M. Dagenais. Tracing for hardware, driver, and binary reverse engineering in linux. *CodeBreakers Jounal*, 1, 2006.

[13] M. Desnoyers and M. R. Dagenais. Low disturbance embedded system tracing with linux trace toolkit next generation. In *Embedded Linux Conference 2006*, 2006.

[14] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. Statl: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10:71–103, 2002.

[15] F. C. Eigler. Problem solving with systemtap. In *Ottawa Linux Symposium*, 2006.

[16] C. LaRosa, L. Xiong, and K. Mandelberg. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing*, 2008.

[17] LTTng. http://lttng.org. Retrieved on 2009-03-10.

[18] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. B. Irving, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. In *IEEE Computer magazine*, 1995.

[19] RAGEL. http://www.complang.org/ragel. Retrieved on 2009-03-07.

[20] G. Vigna, S. T. Eckmann, and R. A. Kemmerer. The stat tool suite. In *DARPA Information Survivability Conference & Exposition*, 2000.

[21] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Lecture Notes in Computer Science*, 2004.