
# PROCEEDINGS OF

# The 2nd ACM SIGOPS Asia-Pacific Workshop on Systems

**July 11-12th, 2011**
**Shanghai, China**

**Sponsors**


acm
SIGOPS
ACM SIG on Operating Systems


Microsoft
Research


USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION


EMC²


Google


Baidu 百度


intel


有道 youdao
网易旗下搜索


NEXR
TOWARD OPEN PLATFORM

# Workshop Organizers

## Steering Committee
Gernot Heiser (NICTA, UNSW, Open Kernel Labs)
Doug Terry (Microsoft Research, Silicon Valley, USA)
Zheng Zhang (Microsoft Research-Asia, China)
Frans Kaashoek (Massachusetts Institute of Technology, USA)
Peter Druschel (Max Planck Institute, Germany)
Mahadev Satyanarayanan (Carnegie Mellon University, USA)

## General Chairs
Haibo Chen (Fudan University, China)
Zheng Zhang (Microsoft Research-Asia, China)

## Program Chairs
Sue Moon (KAIST, Korea)
Yuanyuan Zhou (University of California, San Diego, USA)

## Program Committee
Lorenzo Alvisi (UT-Austin, USA)
Ranjita Bhagwan (Microsoft Research, India)
Ed Chang (Google, China)
Wenguang Chen (Tsinghua University, China)
Gernot Heiser (University of New South Wales, Australia)
Ben Jai (Delta Electronics, Taiwan)
Sam King (University of Illinois, Urbana-Champaign, USA)
Jim Larus (Microsoft Research-Redmond, USA)
Qiong Luo (HKUST, Hongkong)
Sang Lyul Min (SNU, Korea)
Gilles Muller (Inria, France)
Akihiro Nakao (University of Tokyo, Japan)
Dilma Da Silva (IBM TJ Watson, USA)
Lin Tan (University of Waterloo, Canada)
Chandu Thekkath (Microsoft Research-Sillicon Valley, USA)
Geoff Voelker (University of California, San Diego, USA)
Junfeng Yang (Columbia University, USA)
Haifeng Yu (National University of Singapore, Singapore)

## Poster Chair
KyoungSoo Park (KAIST, Korea)

## Web Chairs
Xiao Ma (UCSD/UIUC, USA)
Ding Yuan (UCSD/UIUC, USA)

## External Reviewers

Lintao Zhang (Microsoft Research-Asia, China)
Binbin Chen (National University of Singapore, Singapore)
Chengdu Huang (Pattern Insight, USA)
Weihang Jiang (University of California, San Diego, USA)
Xuezheng Liu (Microsoft Research, USA)
Shan Lu (University of Wisconsin, USA)
Xiao Ma (University of California, San Diego, USA)
Yoann Padioleau (Facebook, USA)
Soyeon Park (University of California, San Diego, USA)
Feng Qin (The Ohio State University, USA)
Yao Shi (NICTA, Australia)
Weiwei Xiong (University of California, San Diego, USA)
Ding Yuan (University of California, San Diego, USA)
Zheng Zhang (Microsoft Research-Asia, China)
Lidong Zhou (Microsoft Research-Asia, China)

## Panelists

Peter Druschel (MPI-SWS)
Gernot Heiser (University of New South Wales)
Jaejin Lee (Seoul National University)
Doug Terry (MSR Silicon Valley)
Geoff Voelker (UC San Diego)
Honesty Young (Intel APAC R&D)
Zheng Zhang (MSR Asia)

# The 2nd ACM SIGOPS Asia-Pacific Workshop on Systems

July 11-12th, 2011
Shanghai, China



## July 11th (Monday)



### Session 1 - Kernel (Chair: Chandu Thekkath)



### Session 2 - Reliability and security (Chair: Gernot Heiser)

### Session 3 - Virtualization (Chair: Geoff Voelker)



## July 12th (Tuesday)



### Session 4 - Cloud computing and data center (Chair: Wenguang Chen)

# Message from the TPC Chairs

Welcome to the 2nd Asia-Pacific Systems Workshop (APSys 2011)! It is our great pleasure to host this exciting event in the fascinatingly modern and yet historical city of Shanghai in China. The first APSys workshop was co-located with the ACM SIGCOMM 2010 Conference held in New Delhi, India. This year we took a bold step toward making APSys an independent event on its own and are holding on in July 11th and 12th, 2011.

The goal of APSys is to build a forum for systems researchers and practitioners to present their work in computer systems and for locals in Asian and Pacific regions to meet, interact, and collaborate with top researchers in the field. This year we have received a record number of 57 submissions and accepted 19 papers for a one-and-a-half day schedule. All papers received 3 to 4 reviews, mostly done by the PC members. We thank all the PC members and external reviewers for their valuable time and contributions. The full program includes two keynote talks by world-renowned Peter Druschel and Kai Li, a poster session organized by KyoungSoo Park, and a panel discussion with the help of Geoff Voelker and Chandu Thekkat.

We have received generous support from our sponsors, ACM SIGOPS, Microsoft Research, USENIX, Google, Baidu, NexR, Intel, Youdao, and EMC and thank them. Our general chairs, Zheng Zhang and Haibo Chen, helped every way to keep all of us on schedule. The steering committee have generously shared their experiences and wisdom to turn APSys 2011 into a great success. Lastly but of course not the least, we would like to thank our web chair, Xiao Ma, whose diligence and promptness kept us on track always. Thank you all and enjoy APSys 2011!

TPC chairs, Sue Moon and Yuanyuan Zhou

# Keynote - Kai Li (Princeton University)

Kai Li is a Paul M. Wythes '55, P'86 and Marcia R. Wythes P'86 Professor at Princeton University, where he worked as a faculty member since 1986. Before joining Princeton University, he received his Ph.D. degree from Yale University, M.S. degree from University of Science and Technology of China, Chinese Academy of Sciences, and B.S. degree from Jilin University. His research expertise is in building parallel and distributed systems, deduplication storage systems, and data analysis and search for large feature-rich datasets. He is an ACM fellow and an IEEE fellow. In 2001, he co-founded Data Domain, Inc., serving in roles as the initial CEO, CTO and Chief Scientist.

# Protected Hard Real-time: The Next Frontier

Bernard Blackham, Yao Shi and Gernot Heiser
NICTA and The University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

## ABSTRACT

Hard real-time systems are typically written to execute either on bare metal or on a small real-time executive that offers no memory protection. This model scales poorly as systems become more complex and integrated, as is the trend in industry today. Designing hard real-time systems on a protected OS is often avoided due to the difficulty in predicting its response time.

Hard real-time systems with full virtual memory and memory protection have been proposed previously. However, to our knowledge, no such system has determined safe upper bounds on the latency introduced by this protection.

This paper proposes that hard real-time systems can be constructed confidently and cost-effectively using an operating system providing full memory protection and virtual memory. We contend that a carefully written microkernel providing these mechanisms has the ability to be used in a hard real-time system without overly pessimistic response time guarantees. We use the seL4 microkernel as a case study, investigating how the features of seL4's design enable a highly accurate WCET analysis.

## Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

## General Terms

Design, Reliability, Performance

## Keywords

Microkernels, worst-case execution time, hard real-time systems, trusted systems

## 1. INTRODUCTION

Traditionally, hard real-time systems are constructed on hardware with predictable timing characteristics and with minimal software "glue" between the application and the hardware itself. Such systems are often developed without an operating system—on "bare metal"—or use a lightweight real-time executive to schedule threads. They offer no memory protection between components. The lack of fault tolerance leads to a design that is difficult to confidently scale to complicated systems which integrate several complex software stacks on one processor.

Large systems often separate out critical real-time functionality onto dedicated processors, such as the baseband processor found on most smart phones. However, as manufacturers strive to gain a competitive advantage by adding features to embedded devices, the level of integration will only increase. Using dedicated processors does not scale—for example, cars and aircraft are trending towards combining both critical and convenience functionality, and the cost and weight of tens or hundreds of processors is a serious issue.

An alternative solution to satisfy this growing trend is to consolidate these systems onto a single processor, and use an operating system to provide isolation between critical real-time components and less critical time-sharing components [MHH02]. However, for hard real-time designs, this solution depends on the ability to provide safe upper bounds on the interrupt latency of the OS. In most systems, the interrupt latency is determined by the maximum worst-case execution time (WCET) of all non-preemptible code in the kernel.

It is possible to achieve very good interrupt latencies by making the kernel fully preemptible. In this model, interrupts are permitted to occur anywhere within the kernel, except within some small protected regions of code, usually to modify critical data structures. This gives typical interrupt latencies in the order of tens, or hundreds, of cycles. However, this requires very careful coding of the interrupt paths, and defensively analysing that at every point in the kernel an interrupt cannot cause a crash or make the kernel's state inconsistent. Analysing concurrency issues of this nature is extremely challenging due to the explosion of possible interleavings to consider and the difficulty in reproducing timing-related bugs. Much research effort has been devoted to developing methods and tools to identify such cases.

Many kernels do not allow interrupts to occur whilst executing kernel code, or allow them only to occur at designated preemption points. This greatly simplifies the design and testing of the kernel, at the cost of higher worst-case interrupt latency. Well-placed preemption points mitigate the issue, but still cannot achieve the small latencies of the fully preemptible model.

As embedded processors become faster, guaranteed latencies in the 100 000s of cycles become acceptable for many applications. This, in turn, permits the integration of larger, more complex software components into a single system, keeping device costs lower. The challenge then for hard real-time systems is to compute a safe

upper bound on the interrupt latency of the kernel.

Safe upper bounds for WCET are generally computed using a combination of static analysis techniques and measurements on real hardware [KWRP05,PZH07]. Operating systems kernels have long been an elusive target of static WCET analysis, due to their unstructured code, tight coupling with hardware, and sheer size. WCET bounds based on measurements alone cannot be relied upon—for example, estimated upper bounds from measurements stated for RTLinux [YB97] were later shown to be invalid [MHSH01]. Several kernels have been analysed using static analysis, including RTEMS [CP01] and OSE [SEGL04, CEE+02], but these did not support memory protection using paged virtual memory. To our knowledge, no kernel providing full virtual memory and memory protection has been successfully analysed for its WCET. An analysis was attempted on the L4 Pistachio kernel [SP07], but safe WCET bounds were never established.

We assert that a well-designed microkernel lends itself to a tight WCET analysis. This paper focuses on the seL4 microkernel and presents it as a viable solution for creating hard real-time systems with very strong isolation properties.

## 1.1 seL4 as a Hard Real-time Platform

seL4 [KEH+09] is a third-generation microkernel, broadly based on the concepts of L4 [Lie96]. It provides virtual address spaces, threads, communication via synchronous and asynchronous IPC, and capabilities for managing authority. The distinguishing feature of seL4 is that it is the only kernel to date with a formal machine-checked proof that the C code implementation adheres to the specification of the kernel. This additionally ensures that seL4 will never crash or perform an unsafe operation. Whilst these strong functional guarantees are sufficient for many systems, critical real-time systems also require temporal guarantees to achieve safety.

seL4's specification dictates that the kernel will never enter an infinite loop—i.e., all seL4 system calls eventually return to the user. Previously, this was the only temporal guarantee known of seL4. In this paper, we investigate seL4's application to hard real-time domains and present the benefits of analysing a formally verified kernel. A microkernel with provably correct operation and guaranteed worst-case execution time bounds creates a foundation on which large-scale, trustworthy, hard real-time systems can be built.

## 1.2 Contribution

This paper asserts that it is possible to compute realistic safe upper bounds on interrupt latency for protected microkernel-based systems. We demonstrate the first full WCET analysis of a memory-protected OS kernel, seL4, with a view to tuning the kernel for hard real-time applications. We perform a full *context-aware* analysis of all of seL4's code paths—specifically, the analysis virtually inlines all functions within seL4 so that it is context-sensitive. Such an approach is feasible due to seL4's small code size (compared with other operating system kernels), at around 8 700 lines of C code. Despite this fact, it is, to our knowledge, still the largest code base where a full context-aware WCET analysis has been performed.

Section 2 details the features of seL4 that make it amenable to automated analysis. Section 3 describes the methods used to analyse seL4. Section 4 shows the results of the analysis, outlining the worst-case execution paths found.

# 2. SEL4 DESIGN FEATURES

The seL4 microkernel has several properties that assist with automated static analysis. First and foremost is that its code base is small. We analysed the ARM version of the seL4 kernel, which has around 8 700 lines of C code and 600 lines of ARM assembly code. Although this is a large body of code by WCET analysis standards, we found it to be just within the scalability limit for the implicit path enumeration technique (IPET) [LMW95]. The full analysis takes two hours to compute, and is described further in Section 3.

seL4 is an event-based kernel, where a single kernel stack is shared by all user threads. Context switching between user threads is performed by changing a variable containing the currently running thread. In contrast, process-based kernels, with dedicated per-thread kernel stacks, must switch the stack pointer during a context switch. This model may be more efficient in the presence of frequent context switches [Lie93b], but the event-based model of seL4 aids static analysis significantly, as control flow is more structured.

Other features that simplified our analysis are listed below. Many of these arose due to requirements of the formal verification process, without any regard to a WCET analysis.

- seL4 never stores function pointers at run-time, so all jumps can be resolved statically (with the help of symbolic execution).
- seL4 never passes pointers to stack variables. This simplifies the analysis of memory aliasing for WCET.
- The task of memory allocation is delegated to userspace, avoiding complex allocation routines within the kernel.
- There are very few nested loops within seL4 – automatically identifying nested loops at the assembly level and their loop relations is not an easy task in the presence of heavy compiler optimisations.
- Unbounded operations (such as object deletion) contain explicit preemption points. If an interrupt is pending at a preemption point, seL4 will postpone the current operation and return to a safe state to handle the interrupt.

seL4 is accompanied by a large body of machine-checked proofs which contains thousands of invariants and lemmas. It should be possible to incorporate these into a WCET analysis to assist in excluding many infeasible paths.

One issue that arose during the analysis of seL4 is that in two places mutually-recursive functions are used. The formal proof guarantees termination and actually proves that the functions do not call themselves more than once. This knowledge makes the analysis easier, as we could simply virtually inline each function at most twice. However, for this analysis, we chose to unwind the recursion manually.

The design of seL4, in conjunction with formally-proven guarantees, has greatly assisted in performing an automated static analysis.

# 3. ANALYSIS METHOD

We performed a static analysis of the seL4 kernel binary to compute a safe upper bound of its WCET. For comparison, we constructed the worst-case scenarios detected by the analysis and executed them on real hardware. This gives a indication of how tight the analysis is. Table 1 summarises the relevant properties of the code analysed.

## 3.1 Static Analysis

We analysed seL4 for its interrupt latency by examining the worst-case execution time of all possible paths through the kernel, accounting for preemption points. Non-preemptible paths can begin at a number of places, such as entry to a system call or page-fault handler. Interrupts can be processed only once control is returned to the user. In seL4, explicit preemption points detect if an interrupt is pending within a long-running loop and if so, postpones the current operation and returns up the call stack. The interrupt latency is

**Table 1: Properties of the analysed seL4 binary.**

| | |
|---|---|
| Code size | 98704 bytes |
| Lines of code | 8642 |
| Number of functions | 84 |
| Number of basic blocks | 1922 |
| Number of loops | 68 |
| Number of branches | 1410 |

the sum of the WCET of the longest kernel path and the time taken to dispatch the interrupt to a user thread.

The seL4 binary we analysed was compiled with gcc using `-O2` optimisation level and the `-fwhole-program` flag, which enables gcc to perform very aggressive optimisation and inlining of code. This means that most function boundaries are lost and functions are on average much larger because of inlining. The compiled binary also exhibits optimisations such as tailcalls and loop rotation.

Despite having well structured code, seL4 violates this structure in one specific code path. seL4 features a highly optimised routine for handling the most common IPC operations, known as the *IPC fastpath*; it improves the average time for these IPC operations by an order of magnitude. It does this using a continuation-based control flow, avoiding the need for stack unwinding. Unfortunately, the analysis tool currently does not support continuations—it expects all functions to return. As a result, we needed to disable the IPC fastpath at compile time. However, we do not expect this to affect our analysis, given the aforementioned presence of order-of-magnitude slower operations elsewhere in the microkernel.

The control flow graph (CFG) of seL4 is extracted from the binary, using symbolic execution to resolve indirect branches (via a register) and jump tables generated by switch statements. This step was performed without any user guidance, made possible by the absence of function pointers in seL4's sources.

The iteration counts of loops were specified by hand. Most have fixed bounds and could have been determined automatically with a rudimentary analysis. Some, however, depend on the state of the system—e.g. the number of runnable threads. These properties are all bounded by total physical memory. To support this we allow the user to provide an expression relating the iteration count to constants such as the size of physical memory. Due to heavy inlining by the compiler, none of the iteration counts in the binary are context-sensitive, even though some are at the source level (e.g. memcpy).

The control flow graph, along with the loop iteration counts, is passed to a modified version of Chronos 4.0, from NUS [LLMR07]. We adapted Chronos to support the ARM processor. Chronos uses the IPET method [LMW95], which converts the control flow graph into a system of linear equations (or inequalities) with integer coefficients. Chronos extends the basic IPET model with support for instruction caches and pipeline modelling. All function calls are virtually inlined so that the analysis is context-aware. This inlining results in almost 400 000 CFG nodes (basic blocks) in the analysis.

The output of Chronos is a system of linear constraints and an objective function to maximise subject to those constraints. With 400 000 CFG nodes, it creates two million variables and 2.5 million equations.

Finally, an off-the-shelf integer linear programming solver is used to compute the final WCET value. We used IBM's ILOG CPLEX Optimizer to compute the solution. This is the most computationally intensive step of the process, and takes up to two hours for the entire seL4 kernel, when performed on an Intel Core 2 Duo running at 2.93 GHz. However, smaller portions of the kernel are solved much faster—typically within a minute or less.

## 3.2 Hardware Measurements

Our test platform for measurements is a Beagleboard-xM with a TI DM3730 processor. This processor has an ARM Cortex-A8 core running at 800 MHz, with a 32 KB L1 instruction cache and a 32 KB L1 data cache, both 4-way set-associative. The experiments were configured to use 128 MB of physical memory. The latency of a read or write to physical memory on this platform was measured to be 80–100 cycles.

The L1 caches on the Cortex-A8 have an unspecified random replacement policy. This makes simulating the exact cache behaviour impossible, and effectively forces any safe cache analysis to assume a direct-mapped 8 KB cache. Furthermore, it makes it infeasible to construct a true worst-case scenario on hardware.

The Cortex-A8 has a dual-issue pipeline, which is not accounted for in our processor model. Whilst it is in theory possible to force the Cortex-A8 to single issue, this oddly requires a "high security" version of the processor which is not readily available. This means that we can expect the observed results to be up to 2x faster than computed by static analysis. Extending the static analysis model to support a dual-issue pipeline is the subject of future work.

The Cortex-A8 also supports speculative prefetching and branch prediction. These features were disabled in order to make measurements more deterministic. This results in a fixed 13-cycle latency on each branch.

Our experiments also disabled the data cache and L2 cache during both estimation and real execution, as our analysis tools do not yet support these on the ARM platform. This allowed us to confidently validate our timing model.

## 3.3 Open vs. Closed Systems

We analyse seL4 for two different use-cases—open systems and closed systems. We define an *open system* to be one where the system designer cannot prevent arbitrary code from executing on the system. This is in contrast to a *closed system*, where the system designer has full control over all code that executes.

In an open system, real-time subsystems may execute in conjunction with arbitrary and untrusted code (although confined by the capabilities provided to them). seL4 uses a strict priority-based round-robin scheduler. In such a scheme, time sensitive threads must be assigned the highest priority on the system so that they may run as soon as required (typically when triggered by a hardware interrupt). seL4's design disables interrupts whenever in the kernel, except at a few select preemption points. As a result, the interrupt latency for the highest-priority thread is determined by the worst-case execution time of all possible operations performed by seL4.

In a closed system, the system designer has full control over all operations performed by the kernel. Therefore she can ensure that operations that are known to be long-running do not occur at critical times, e.g. by allocating all resources at boot time and avoiding delete operations at run time. The interrupt latency in this scenario is defined by the WCET of a select number of paths within the kernel which are used by the running system—primarily inter-process communication (IPC) operations, as well as thread scheduling. The permitted system calls are listed in Table 2.

Note that `seL4_Call()` can be invoked on an IPC object to perform IPC operations, but invoking it on other object types may lead to the creation or deletion of kernel objects. We exclude these latter operations from the analysis of closed systems, allowing only the IPC-related uses of `seL4_Call()`.

**Table 2: System calls permitted in a closed system.**

| System call | Description |
|---|---|
| `seL4_Send()` | Blocking send to an endpoint. |
| `seL4_Wait()` | Blocking receive on an endpoint. |
| `seL4_Call()` | Combined blocking send/receive. |
| `seL4_NBSend()` | Non-blocking send to an endpoint (fails if remote is not ready). |
| `seL4_Reply()` | Non-blocking send to most recent caller. |
| `seL4_ReplyWait()` | Combined reply and wait. |
| `seL4_Notify()` | Non-blocking send of a one-word message. |
| `seL4_Yield()` | Donate remaining timeslice to a thread of the same priority. |

**Table 3: Computed upper bound versus measured observations for feasible worst-case paths with data caches disabled.**

| Case | Computed | Observed | Ratio |
|---|---|---|---|
| Endpoint deletion | 686.0 ms | 155.1 ms | 4.42 |
| IPC (open system) | 635.1 ms | 272.8 ms | 2.33 |
| IPC (closed system) | 1148.2 $\mu$s | 118.2 $\mu$s | 9.71 |

# 4. EXPERIMENTAL RESULTS

## 4.1 Open System

In an open system, the analysis pointed us to two interesting cases which were clear candidates for the worst-case execution path in seL4.

The first case arises due to the nature of IPC in seL4. Threads do not communicate with each other directly. Rather, they construct IPC "endpoints" which act as communication channels between threads. Multiple threads are permitted to wait to receive (or send) a message on a single endpoint—threads join a queue and are woken in turn as partners arrive. If the endpoint is deleted whilst there are still multiple threads waiting, each of these threads is removed from the endpoint queue and added to the scheduler's run queue. A malicious program (looking to force a deadline miss), could allocate as many threads as possible and construct this scenario. We constructed such a scenario with 91 000 threads (limited by physical memory). The results are shown in Table 3.

The second case arises due to a scheduler optimisation used in seL4 known as lazy scheduling [Lie93b]. In microkernel-based systems where IPC is frequent, a thread blocking on an IPC operation will often be made runnable again before the scheduler even needs to reconsider it for execution. To benefit from this observation, seL4 does not immediately remove threads from the run queue, but defers that work until a thread is selected to be scheduled. This leads to the obvious worst-case scenario where many non-runnable threads pollute the run queue. The scheduler must iterate over all of these threads, inspect and then dequeue them, until it finally finds a runnable thread (or the idle thread).

We constructed this scenario, using the `seL4_TCB_Suspend()` operation which suspends a thread but does not immediately dequeue it from the run queue. The second row of Table 3 compares our computed value with measurements observed on hardware. In this case, a system with 128 MB memory can create 119 720 threads.

## 4.2 Closed System

Within a closed system, where only the system calls outlined earlier in Table 2 are permitted, our analysis detects an infeasible worst-case scenario. The `seL4_Reply()` operation is used to respond to the most recent message received with `seL4_Wait()`. A one-time endpoint used to respond to the most recent sender (known as a *reply cap*) is stored in a dedicated location in each thread control block (the *reply slot*). The kernel must delete the existing reply cap before any call to `seL4_Wait()` and after a call to `seL4_Reply()`.

The analysis detected that deleting this reply cap could lead to a long delay at the next reschedule, for the same reasons as outlined in the first scenario of the open system, described earlier. Even though we excluded explicit delete operations from our analysis, this implicit operation was exposed. However, it is impossible to construct this scenario, as reply caps can only be used by other threads if they are first removed from the reply slot. Therefore the delete operation on the reply slot will only affect the schedulability of one thread.

With this knowledge, we could add an extra constraint which excluded this infeasible path. The new analysis determined that all IPC send or receive operations became candidates for the new worst-case path. It identified two factors which affect the IPC operation's execution time. The first is that endpoints are addressed using a structure resembling guarded page tables [Lie93a]; decoding the address involves traversing a graph up to 32 levels deep. The second is, unsurprisingly, the size of the message to be transferred, on which seL4 places a hard limit of 120 32-bit words. The combination bounds the worst-case interrupt latency of a closed system to a very reasonable limit.

This case was also reproduced on hardware and measured, using the `seL4_ReplyWait()` system call. The results are shown in the final row of Table 3.

## 4.3 Analysis of Results

Table 3 shows that there is a factor of up to 9.71x between the observed and computed execution times. This disparity can be attributed to both the random cache replacement policy of the instruction cache, as well as the dual-issue pipeline of the Cortex-A8. With a random cache replacement policy, constructing a true worst case on hardware is extremely difficult. Modelling the Cortex-A8 pipeline perfectly is also a difficult task. Given that the memory access latency on fast processors far outweighs the impact of pipeline effects, a simpler pessimistic pipeline model is sufficient. None of these factors cause the static analysis to be unsound and therefore the computed values, though large, can be confidently used as a safe upper bound for hard real-time systems.

It should be noted that these results are much worse than reality as the data cache has been disabled both on hardware and in the model. As memory latency is up to 100 cycles on this platform, this adds a significant factor to the execution time of these test cases.

Certain code paths are guaranteed by the formal proof never to execute. These paths could potentially be pruned by incorporating invariants from this proof into the WCET analysis. For example, there is an existing invariant in the seL4 proof which specifies that the reply slot may contain a reply cap or is otherwise empty—no other type of capability can reside there. This specific invariant eliminates the infeasible path described in Section 4.2.

# 5. CONCLUSIONS AND FUTURE WORK

As the trend of feature integration in embedded devices continues to gain momentum, integrating numerous complex software stacks in a fault-tolerant manner will be a necessity. In this paper, we assert that microkernels can be used as the basis for hard real-time systems that nonetheless feature such integration. The primary requirement placed on these microkernels is a reasonable

guarantee on their interrupt latencies.

A tight static analysis of a microkernel to determine safe WCET bounds is in fact feasible, as demonstrated by our analysis of seL4. There are many features of seL4 that both ease the analysis process and reduce the interrupt latency, without the need for a fully preemptible kernel.

For the feasible paths in seL4, the disparity between our calculations and measurements arises for two reasons: first, the nondeterminism of the target hardware, and second, surmountable limitations of our analysis tools.

Future work will focus on adding support for the data cache, and automatically incorporating proof invariants into the WCET analysis to further tighten the computed upper bound.

Concurrently, seL4 development will be guided by the results from the WCET analysis. It is clear that lazy scheduling is not a suitable optimisation for a real-time kernel and alternative methods should be investigated. The analysis has also determined areas in seL4 where preemption points should be added to bound interrupt latency.

At present, seL4 can be used in a closed system with reasonably small guaranteed response times. Many applications, such as deeply-embedded systems, are consistent with the closed system model. In an open system, allowing untrusted code to execute, the response time guarantees are still bounded but too large to be useful. These results highlight areas where seL4's real-time behaviour can be improved.

## 6. ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

## 7. REFERENCES

[CEE+02] Martin Carlsson, Jakob Engblom, Andreas Ermedahl, Jan Lindblad, and Björn Lisper. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *2nd International Workshop on Real-Time Tools*, 2002.

[CP01] Antoine Colin and Isabelle Puaut. Worst case execution time analysis of the RTEMS real-time operating system. In *13th ECRTS*, pages 191–198, Delft, Netherlands, Jun 13–15 2001.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KWRP05] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using measurements as a complement to static worst-case execution time analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec 2005.

[Lie93a] Jochen Liedtke. A high resolution MMU for the realization of huge fine-grained address spaces and user level mapping. Arbeitspapiere der GMD No. 791, German National Research Center for Computer Science (GMD), Sankt Augustin, Germany, 1993.

[Lie93b] Jochen Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Asheville, NC, USA, Dec 1993.

[Lie96] Jochen Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. In *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, volume 69(1-3), Dec 2007.

[LMW95] Yau-Tsun Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[MHH02] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *23rd RTSS*, Austin, TX, USA, 2002.

[MHSH01] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. RTLinux with address spaces. In *3rd Real-Time Linux WS*, Milano, Italy, nov 2001.

[PZH07] Stefan M. Petters, Patryk Zadarnowski, and Gernot Heiser. Measurements or static analysis or both? In *7th WS Worst-Case Execution-Time Analysis*, Pisa, Italy, Jul 2007. Satellite WS 19th ECRTS.

[SEGL04] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, October 2004.

[SP07] Mohit Singal and Stefan M. Petters. Issues in analysing L4 for its WCET. In *1st MIKES*, Sydney, Australia, Jan 2007. NICTA.

[YB97] Victor Yodaiken and Michael Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997.

# Our Troubles with Linux and Why You Should Care

Ashif S. Harji Peter A. Buhr Tim Brecht
Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
asharji,pabuhr,brecht@uwaterloo.ca

## Abstract

Linux provides researchers with a full-fledged operating system that is widely used and open source. However, due to its complexity and rapid development, care should be exercised when using Linux for performance experiments, especially in systems research. The size and continual evolution of the Linux code-base makes it difficult to understand, and as a result, decipher and explain the reasons for performance improvements. In addition, the rapid kernel development cycle means that experimental results can be viewed as out of date, or meaningless, very quickly. We demonstrate that this viewpoint is incorrect because kernel changes can and have introduced both bugs and performance degradations.

This paper describes some of our experiences using the Linux kernel as a platform for conducting performance evaluations and some performance regressions we have found. Our results show, these performance regressions can be serious (e.g., repeating identical experiments results in large variability in results) and long lived despite having a large negative effect on performance (one problem has existed for more than 3 years). Based on these experiences, we argue: it is sometimes reasonable to use an older kernel version, experimental results need careful analysis to explain why a performance effect occurs, and publishing papers validating prior research is essential.

## Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Distributed Systems—*client/server*; D.4 [**Operating Systems**]: Reliability—*verification*; D.4 [**Operating Systems**]: Performance—*measurements*

## General Terms

Experimentation, Measurement, Performance

## Keywords

Linux, bugs, web servers, regression testing


*2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)* July, 2011, Shanghai, China


## 1. INTRODUCTION

Linux is a boon to academic systems-researchers because it provides an open-source platform to make improvements to and evaluate the performance of systems that are used in production. Prior to Linux, OS, networking and database researchers were often at the mercy of OS vendors with respect to developing and evaluating new OS mechanisms or policies. Furthermore, researchers could only find bugs or performance problems in a vendor's OS by treating it as a black box and developing external tests. As well, vendors were frequently unreceptive to performance problems and bug reports. In fact, the availability of Linux has forced some OS vendors to make some or all their software open source, allowing researchers alternative venues for development. Fundamentally, without access to source code, it is extremely difficult for researchers to innovate in the crucial and expanding area of systems software. Jockeying for special or restricted access to an OS, or working with the OS as a black-box does not allow all researchers equal or sufficient access to find, understand and fix logic or performance problems.

However, due to its complexity and rapid development, using Linux for systems research also has its drawbacks. The complexity of a large software system makes it difficult to configure and tune for the best possible performance, and it also makes understanding and explaining the results difficult. In addition, the rapid kernel development cycle means that experimental results can be viewed as out of date, or meaningless, very quickly. But most importantly, the rapid changes in kernel development introduce both bugs and performance degradations. While bugs causing failures are quickly identified and fixed, performance related problems are extremely difficult to isolate and correct. Furthermore, because of conflicting goals and tradeoffs that are central to systems implementation, changes that increase performance in one area may degrade performance in another.

The main contribution of this paper is demonstrating that significant performance problems exist in multiple Linux kernels. As well, we argue that:

- some performance results published over an extended time-period need to be re-examined due these performance issues.
- to encourage good science, publishing papers that validate prior research results is essential.
- experimental results need careful analysis to understand and explain why a change has or has not produced a performance effect.

- finding and fixing performance problems is difficult and time consuming, as is getting performance fixes into the Linux kernel.
- changing to the newest Linux kernel is neither a panacea nor requirement for sound research.

Finally, we make a number of recommendations for performing sound experimental performance evaluations.

## 2. EXPERIENCES WITH BUGS

We conduct research into designing and testing web-server architectures on uniprocessor and multiprocessor hardware with the goal of understanding how differences in architecture affect performance. During detailed comparisons of various servers, a number of performance anomalies were encountered that could not be explained based on server architectures or configurations [1]. The anomalies were eventually tracked into the Linux kernel, where three Linux performance problems were found. These problems are subtle as they do not cause crashes or typically result in crippling performance behaviour. Without the benefit of working with multiple servers and access to the Linux source-code, it would have been difficult to identify these anomalies.

### 2.1 Small File Evictions

*Kernel versions affected: 2.6.11 to 2.6.21.7*
*Duration: 02-Mar-2005 to 04-Aug-2007*

There was a bug where small files ($\leq$ page size) were being evicted from the file-system cache regardless of their frequency of access. The bug occurred when a change was made to the file-system cache-code to prevent a single, sequential non-page-aligned read of a large file from invalidating a large portion of the file-system cache. However, the mechanism used to detect this behaviour was too coarse; multiple consecutive accesses to the same page in the file-system cache did not update the access flags for that page. Only when a different page in the file is accessed are the access flags updated. This logic results in small files never being marked as accessed after their first access. Hence, these pages are always evicted from the cache regardless of how often the file is accessed.

Situations where the file-system cache fills and must begin evicting pages to disk are potentially affected by this problem. The problem manifests itself through poor disk performance because of less efficient disk access resulting from small, frequently accessed files constantly being reread from disk as opposed to sitting in the cache. The problem becomes acute for applications that place a heavy load on the file-system cache, e.g., web servers, particularly when small files constitute a significant portion of the workload.

The small-file-evictions problem was discovered after publishing performance results [4] using a kernel that contains this bug. The bug was found while conducting subsequent research using a different workload with increased disk I/O. After finding the bug, we reexamined the experiments from the paper and fortunately determined it had only a minor effect on the results, but an effect nonetheless. Hence, we were lucky and the conclusions in the paper are still valid.

### 2.2 Prefetching Disabled

*Kernel versions affected: 2.6.12 to 2.6.22.19*
*Duration: 17-Jun-2005 to 26-Feb-2008*

There was a bug where the page-cache read-ahead is disabled for sequential disk-reads when using sendfile with non-blocking sockets, as a result of the kernel misinterpreting the access pattern when reading large files. sendfile is unusual because a call can involve both disk and network I/O. Multiple sendfile calls may be necessary to transmit file data over the network because the size of non-blocking sends are limited by the socket-buffer size. Similarly, the operating system reads a file into the file-system cache from disk in pieces, with the size of each piece determined by the disk-I/O scheduling algorithm. For large files requiring disk-I/O (i.e., not already in the file-system cache), the socket-buffer size is normally smaller than the amount of file-data read by a single disk-request, so the number of disk accesses required is fewer than the number of network transmissions for the send.

As a result, for nonblocking sendfile, the file-access pattern appears random because consecutive sendfile calls, when transmitting a large file, do not appear to continue from the end of the last disk I/O but rather continue from some location *within* the last disk read. At this point, the kernel disables page-caching read-ahead for the file and the size of future disk requests for that file become smaller on average. In contrast, for blocking sendfile, only a single sendfile call is required, and since the kernel performs the appropriate tracking, it recognizes file access is sequential, resulting in correct page-cache read-ahead behaviour. We believe this bug resulted from using or adapting a pre-existing kernel function for use with sendfile that originally simply read file data from disk. Unfortunately, assumptions about what constituted sequential access were not correspondingly adapted to recognize the unusual disk access-patterns resulting from sendfile with non-blocking sockets, causing read-ahead to be disabled. This bug was found while trying to understand and explain the differences in performance obtained when using blocking and non-blocking sendfile.

### 2.3 Erratic Page Evictions

*Kernel versions affected: 2.6.23 until at least 2.6.36.2*
*Duration: 09-Oct-2007 to at least 09-Dec-2010*

There may exist a sendfile bug in the most recent Linux kernel (present the last time we checked in December 2010). This bug results in none of the pages associated with a transmitted file being marked as accessed by the kernel, so the kernel cannot distinguish between recently or frequently accessed pages and other pages in the file-system cache. Therefore, under memory pressure, the kernel may incorrectly evict pages from the file-system cache. When these pages are in the middle of files or frequently accessed, it hampers long contiguous disk-reads and read-ahead buffering, which results in smaller and more random disk requests. This behaviour is manifested as erratic server performance and low disk-throughput.

To correctly mark page accesses for sendfile, we developed a patch for the 2.6.24.3 Linux kernel.[1] This patch provided consistent and repeatable performance measurements, by increasing file-system cache hit rates and improv-

[1]Our patch for small-file evictions (1st bug) is still in place but the code path for sendfile changed significantly from the earlier to the later kernels (due to splice). As a result, our prior knowledge about sendfile could not be used with respect to the new problem.

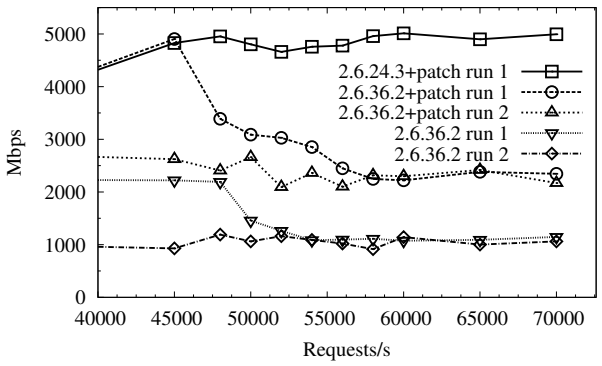


**Figure 1: Throughput for patched and unpatched kernels**

ing throughput when reading files from disk. Disk throughput is increased in some of our experiments from approximately 11,000 blocks-in per second (1 block = 1024 bytes) for non-blocking sendfile and 20,000 blocks-in per second for blocking sendfile to approximately 28,000–30,000 blocks-in per second for *both* non-blocking and blocking sendfile. Figure 1 shows some representative experiments to illustrate this I/O problem in recent kernels. The patched 2.6.24.3 kernel (line 2.6.24.3+patch run 1) has stable performance and the highest throughput. The unpatched 2.6.36.2 kernel (line 2.6.36.2 run 1) has significantly lower throughput and throughput drops substantially for higher request rates. Repeating the experiment a second time (line 2.6.36.2 run 2) shows a large difference in performance at 45,000 and 48,000 requests per second. After applying our patch for the 2.6.24.3 kernel to the 2.6.36.2 kernel (line 2.6.36.2+patch run 1), throughput is significantly higher compared to the unpatched counterparts. However, throughput is still significantly lower than the patched 2.6.24.3 kernel at higher request rates. Even with the patch, a second run (line 2.6.36.2+patch run 2) has different throughput, showing large variations similar to the unpatched kernel. Without the patch, the 2.6.24.3 kernel exhibited similar problems (not shown in the graph). These experiments indicate there may be an additional performance regression with the newer kernel or that the code has changed enough to make the patch less effective. The variance in throughput for identical experiments, combined with low throughput, directed us to investigate this anomaly further and lead to the bug.

As a result of these performance problems, for our research work we either: use the patched 2.6.24.3 kernel because of our experience with this kernel, its stability after patching, and the problems and uncertainty with the newer kernels; or are trying FreeBSD on one new project.

## 3. EXPERIENCES FINDING BUGS

Debugging performance problems is difficult, especially tracking a performance problem into the Linux kernel. Sometimes, the most difficult step is to recognize that a performance problem actually exists. In isolation, it is difficult to determine if an application is running reasonably or if there is a problem with its performance. In our work, we had the benefit of comparing the throughput of several web-servers across various configurations and workloads allowing us to identify performance anomalies. Finding the source of a performance problem can be challenging as problems often occur only when the application is under full load, when debugging and profiling tools may significantly perturb the environment.

Two common tools for tracking bugs in the kernel are OProfile and SystemTap. OProfile generates dynamic call-graphs along with the execution time spent in each function. We found OProfile was not very helpful because it tended to be too coarse grained. Rather, we found tools such as vmstat and mpstat to be more helpful for our particular web-server work. Unexpected differences in their statistics helped to confirm a problem and even suggested the type of problem. SystemTap was used to track down the read-ahead problem with non-blocking sendfile, and helpful with the other problems. It is a scripting language useful for instrumenting a running Linux kernel by executing a handler on specified events, such as on entry to or exit from specified kernel functions, allowing the printing of local data. Without a tool like SystemTap to trace the sendfile call and narrow the search space, finding these problems would have taken significantly longer because the Linux kernel is large and complicated.

## 4. EXPERIENCES OF OTHERS

Some web sites contain data that tracks the performance of different benchmarks over time (in some cases by kernel version) [3, 2]. Browsing through the collections of benchmarks available on these sites examples of long and short term performance regressions and improvements can be found. Specifically, the web site "Linux Kernel Performance!" [3] has tracked the performance of several benchmarks executing on Linux kernels from version 2.6.22 to 2.6.38 (at the time of writing). An example of a short-term performance-regression occurs for the Online Transaction Processing benchmark (OLTP) on a 4P quad-core Xeon. Performance drops by approximately 45% from kernel version 2.6.22 to 2.6.23 and improves in subsequent releases until it is back to the 2.6.22 level in version 2.6.25. An example of a longer-term performance-reduction occurs for the benchmark fileio-cfq on a 4P quad-core Xeon. Performance drops by about 30% from kernel version 2.6.31 to 2.6.32 and performance of this benchmark has not improved from that level with subsequent releases of the kernel (up to 2.6.38).

Interestingly, changing to a 2P Quad-core Core 2 Duo for the same two benchmarks on the two kernel releases (OLTP on 2.6.23 and fileio-cfq on 2.6.32) generates different performance regressions. The degradations are about 20% for OLTP (45% on the 4P system) and only 5% for fileio-cfq (30% on the 4P system). If a regression test is performed on the 2P system, the 5% reduction may be deemed acceptable, but if performed on the 4P system, the 30% reduction may be deemed unacceptable. Furthermore, if the range of kernels is altered to 2.6.29 and 2.6.31, there is a 20% reduction on the 2P, but a 3% increase on the 4p. Therefore, it is necessary to track performance across a number kernel versions on different systems to fully understand performance changes.

Some of the benchmarks exhibit huge swings in performance. For example, on the 4P quad-core Xeon system the benchmark hackbenchpth150 improved by about 2,000% from kernel version 2.6.25 to 2.6.26. Unfortunately, those

gains disappeared with the release of 2.6.36 and have stayed at the reduced level (to version 2.6.38).

Performance regressions cause problems not only for researchers but also for companies. Companies want to use these kernels in production environments to conduct and report results of important benchmarks using a version of the kernel without performance problems.

# 5. CONSEQUENCES

The performance issues raised in the previous sections imply a number of consequences for researchers:

## 5.1 Problems in Published Papers

A number of papers across many disciplines over multiple years may contain incorrect performance results. Based on our experience, papers involving significant I/O may be affected. As well, based on the benchmarking results across different kernel versions, performance variations occurred across different parts of the kernel, so the scope of affected papers/results could be larger than what we report on. The scientific approach to finding incorrect results is for other researchers to reproduce results. Unfortunately, if the original results are verified, it is currently difficult or impossible to publish this work, making the endeavour risky. As in other scientific fields, Computer Science needs to value and publish papers that verify previous results.

## 5.2 Underlying Cause

Based on our experience, it is crucial to find the underlying cause for performance results. Experimental results require careful analysis to understand why a change has or has not produced a performance effect, and anomalies in performance results cannot be ignored because they may be "shouting out" that there is an underlying problem. Determining and explaining the root cause for performance results are likely to lead to either an understanding of the observed performance or the discovery of a problem (in some cases, possibly with the kernel). Simply reporting performance results (either positive or negative) is insufficient.

## 5.3 Fixing Problems

If unexplainable behaviour suggests a bug, it may be necessary to look into the Linux kernel. Our experience is that finding and fixing a kernel bug is extremely difficult and time consuming, especially because the Linux code-base is large and a quickly moving target. For example, there are many levels of indirection (routine pointers) used in the kernel, so determining what is called and when is difficult. Also, the tool-set for monitoring dynamic execution is low-level and complex to use.

Assuming you find and fix a problem, the next logical step is to have the fix applied to the mainline kernel for the benefit of all. Because the kernel evolves rapidly, it is necessary to obtain the most recent kernel and check if the bug is already fixed. If the bug is still present, it may be necessary to port the fix (again) to the new code base. When the code base has changed significantly, it may be the case that people no longer possess the expertise or time required to construct a new fix. Finally, to create a bug report it is important to write small, stand-alone programs that reproduced the problem, and to submit these programs along with the suggested bug fix. Our experience is that bug reports sent to the kernel-developer mailing-list are not always well received and getting our fixes into the mainstream kernel sometimes required a thick-skin and persistence.

## 5.4 Kernel Upgrading Problems

Once your research team has established a working kernel, which generates good, explainable, consistent results, there is the dilemma of moving to the latest version of the kernel because there is a general belief the latest kernel is always better. For researchers, this prejudice appears in the form of reviewers stating that results are not meaningful because the latest kernel is not used. However, based on our experience, bugs we found were not fixed in the new kernel, and new kernels can introduce performance regressions and new problems. Furthermore, new kernels require rerunning and re-validating experiments to re-establish results and gain expertise with the new kernel, which may take weeks or months, and in the meantime another kernel is released. An important aspect of our work has become explaining and justifying why we are using an older kernel.

We expect that other researchers may have similar experiences. Clearly, progress in the Linux kernel is essential, and the people involved are working actively to do the right things. Additionally, there are cases where switching to the newest kernel is absolutely necessary. However, we do want reviewers, kernel developers, system administrators and users to understand that the latest kernel is not always the best kernel. It is incumbent on all parties to clearly state why an old kernel is better than a new kernel or vice versa. The reason needs to be particular and specific, and not just that the new kernel has fixed a number of bugs and improved performance.

# 6. POSSIBLE RECOMMENDATIONS

Linux kernel developers must employ a systematic, sustained regiment of performance regression testing (to our knowledge this is not currently being done). We understand the difficulties in such an undertaking but expect many of the problems we point out could have been avoided had rigorous performance regression testing been an integrated part of the kernel-development process.

Some questions researchers need to ask are: When starting a new project, what version of the kernel should be used and why? When working on a project over an extended period, should the kernel be upgraded and why? If upgrading to a new kernel during a project, does the upgrade change the results significantly, and if so why? How can performance changes be explained by the research that has been done?

Here are some practical suggestions to help answer these questions:

1. Before selecting a kernel, check web sites publishing benchmarks on different kernels and select a kernel with good benchmarks in your research area and avoid those with obvious defects. For example, for reasons explained in Section 4, it is unwise to use version 2.6.23 for workloads that resemble OLTP.

2. After upgrading kernels, run some sanity checks for comparison. If performance improves or degrades, try to determine why. This requires expertise, determination, and time, with no guarantee of success.

3. In general, experiments must be run multiple times to check for variability. If there is variability, explain why and report confidence intervals.

4. When conducting experiments, appeal to your intuition. Researchers sometimes become blind when it comes to obtaining results. If results are significantly better or worse than expected, figure out why.

5. Ensure the experimental environment is sound. For example: address-space randomization (computer security technique) may cause variations in results; Security-Enhanced Linux (SELinux) may reduce performance for some workloads because of its security checking; processor dynamic-frequency-scaling may cause variations in results due to changes in clock frequency. Therefore, it may be appropriate to enable or disable some of these mechanisms depending on the particular experiment.

## 7. SUMMARY

Linux is an excellent platform for both conducting research and as a production environments. Furthermore, the Linux kernel developers are doing an excellent job building an innovative and robust operating-system. This paper would not have been written without their dedication and effort. Working in conjunction with the kernel developers are university and industrial researchers who use Linux to demonstrate new ideas, approaches, and techniques across a spectrum of disciplines. A goal of these researchers is to see their technologies move from the laboratory into production. One reason for Linux's success is its continuous inclusion of innovations and improvements resulting in frequent release cycles.

This paper highlights the issues and the problems that result as the kernel evolves. It is unavoidable that changes must be made to fix bugs, add new features, enhance maintainability, improve scalability, or increase performance. In many cases, kernel developers must make complex decisions regarding tradeoffs among these changes, which can affect different benchmarks and applications in different ways on different systems. Coupled with the relentless pace of change, bugs and performance regressions can occur in newer versions of the kernel.

Our key points are: 1) The latest version of Linux is not necessarily the best version to be using, and researchers, reviewers, kernel developers, and users need to think through and understand the pros and cons of different kernel versions. It may also be the case that the most recent version of the kernel is the best version to be using. 2) In light of the significant performance bugs we found and the time periods over which they have been present, it is very likely that performance results published across an extended time period need to be reevaluated. 3) More papers need to provide a deep analysis of their experimental results. While such analysis is time consuming and difficult, it provides understanding of where the benefits come from and insights into applicability beyond the scope of the paper. 4) The computer-systems research-community needs to embrace the scientific approach of publishing papers that reexamine previous work (in non trivial ways) to either confirm or refute their results. This effort should include different hardware configurations, different operating systems, and different workloads.

## 8. REFERENCES


[1] A. S. Harji. *Performance Comparison of Uniprocessor and Multiprocessor Web Server Architectures*. PhD thesis, University of Waterloo, 2010. http://hdl.handle.net/10012/5040.

[2] M. Larabel. *Five Years Of Linux Kernel Benchmarks: 2.6.12 Through 2.6.37*. Phoronix Media, Nov. 2010. http://www.phoronix.com/scan.php?page=article-&item=linux_2612_2637&num=1.

[3] Linux kernel performance! http://kernel-perf.sourceforge.net.

[4] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the performance of web server architectures. In *Proc. of the 2nd ACM SIGOPS/EuroSys Conf. on Computer Systems*, pages 231–243. ACM, Mar. 2007.

# CertiKOS: A Certified Kernel for Secure Cloud Computing

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, David Costanzo
Yale University
{liang.gu, alexander.vaynberg, bryan.ford, zhong.shao, david.costanzo}@yale.edu

## Abstract

Though attractive as a model for elastic on-demand service, cloud computing solutions based on existing hypervisors cannot guarantee that the provider will service a user's requests correctly, and will not leak sensitive information to unauthorized parties. We introduce **CertiKOS** (**Certi**fied **K**it **O**perating **S**ystem), a hypervisor architecture that leverages formal certification to ensure correctness and counter information leakage in cloud computing. CertiKOS isolates guest applications not only from each other but from provider-controlled resource management mechanisms. The kernel's API gives untrusted, provider-supplied management software control over allocation and delegation of resources such as memory and I/O devices, but prohibits management code from accessing a guest's memory or other resources while in use, or from interfering with a guest's execution except through clean resource revocation. CertiKOS represents an effort to apply recent advances in certified software design to a ground-up design of a modular and evolvable certified kernel. Through machine-checkable proof certificates and runtime monitoring, CertiKOS aims to offer users the assurance of correct and leak-free execution of their cloud services.

***Categories and Subject Descriptors*** D.4.6 [*Operating Systems*]: Security and Protection; D.2.4 [*Software Engineering*]: Software/Program Verification

***General Terms*** Design, Security, Verification

## 1. Introduction

Cloud computing offers a popular model for providers to deploy computing infrastructure and applications on-demand. Running sensitive applications or storing private information in the cloud also poses serious security concerns, however [6]. The risk of information leakage hampers adoption of the cloud model. In addition to external threats such as attacks from other customers on a multi-tenant platform [19], cloud servers also face internal attackers. A malicious operator with access to the provider's management software, or malware targeting hypervisor vulnerabilities [26, 7], can take control of the hypervisor to inspect or manipulate hosted guest computations. Thus, current cloud architectures require users to place high and perhaps unwarranted levels of trust in the operators and hypervisor software responsible for hosting their services.


*APSys'11* July 11–12, 2011, Shanghai, China


We propose a proof-of-concept certified kernel, CertiKOS, as a new trusted foundation for cloud computing. CertiKOS isolates guest services not only from each other, but also from the provider's resource management facilities. In current hypervisor architectures, the management facility is a trusted operating system controlled by the provider—e.g., Xen's Dom0—which has complete access to the private information of any guest. Thus a compromised or a malicious provider can violate both the correctness and privacy of a guest computation. CertiKOS treats the provider's resource management facility as just another client, however. The management facility has the special privileges necessary to set resource management policies and delegate resources, but not to access those delegated resources while in use by clients. To guarantee that CertiKOS correctly implements this isolation, CertiKOS itself will be certified using modern formal tools to satisfy both correctness and information security properties. By using a machine-checkable formal proof to certify a kernel's correctness, together with standard TPM hardware attesting that the provider's system is actually running a formally certified version of CertiKOS, the provider's clients can gain high confidence of the correctness and security of their computations and information in the cloud.

CertiKOS represents a minimalistic design driven by the resource requirements of applications in a cloud environment. The current version of CertiKOS handles the delegation only of CPU cores, RAM, disk, and network I/O. CertiKOS offers privacy guarantees for each of these types of delegated resources. Through hardware-based virtualization [3, 1], CertiKOS also supports transparent isolation and resource management, supporting legacy OSes as clients.

The CertiKOS design leverages recent advances in certified software [23]. We organize CertiKOS as a set of certified modules, applying domain-specific logics and OCAP [10] to link separately certified modules into a complete certified kernel. As CertiKOS handles only resource delegation and isolation, it can be made small and thus feasible for certification. CertiKOS is composed of basic modules such as process management, memory management, network and disk I/O, and isolation and delegation components.

This paper makes the following contributions. First, we present a new architecture for countering information leakage in a cloud computing environment, particularly for defending against malicious service providers. Second, by separating resource delegation from resource management, CertiKOS treats the resource management facility as an untrusted client applications, preventing management software from accessing client spaces. Third, CertiKOS handles the main types of resources present in cloud environments, and can be easily extended to support flexible resource usage polices and provider business models. Fourth, CertiKOS introduces the first kernel architecture designed from the ground up for modular certification, Most importantly, CertiKOS enables end users to share the benefits of cloud computing while gaining confidence that their information will not be leaked or tampered with.

## 2. Motivation and Related Work

Companies are increasingly turning toward cloud providers, such as Amazon EC2, Salesforce CRM, and Rackspace, to host their information, applications, and IT services. In the current architecture, cloud service providers and their data center operators have full control over the machines hosting customers' information and applications, and can access customers' transactions and data at will. Service providers offer informal assurances that they will protect the integrity and privacy of customers' information and computations, but customers have limited means of ensuring that providers actually offer the claimed levels of security. Even assuming the provider and data center operators are normally trustworthy, external attackers may mount attacks from other hosted computations [19]. Worse, internal attackers or malware targeting hypervisor vulnerabilities [26, 7] can take advantage of the provider's management facility to access client information or tamper with guest computations. The cloud computing model would clearly benefit from a mechanism to protect clients' information and computations both from each other and from the management facility, while preserving the advantages of elastic, on-demand resource provisioning that make cloud environments attractive.

***Related Work*** The security of hypervisors and virtual machine monitors is a well-researched topic [21, 25, 4], but traditional hypervisors cannot protect clients from cloud service providers. A hardware-based Trusted Platform Module (TPM) [24] can attest that a machine is running a particular trusted cloud kernel or hypervisor [12, 22], but the size and complexity of modern hypervisors makes it difficult to guarantee and verify their correctness.

Exokernel [9] supports user-level management of physical resources, but is not concerned with information security, and leaks extensive resource management/allocation information to all user space code. NoHype [14] extends the CPU architecture, removing the hypervisor layer to protect a guest OS, but is not applicable to current commodity processors.

Information Flow Control (IFC) enables the explicit labeling of information and controls its propagation. Asbestos [8] and HiStar [28] are operating systems employing IFC to enforce precise information security policies. These kernels are not formally certified, however, and must simply be trusted to enforce information flow control correctly. Additionally, these operating systems do not address threats such as timing side-channels, which represent increasing risks in cloud environments [19].

Singularity [13] implements its base kernel in a typesafe language, but its semantic correctness remains merely an assumption. The seL4 project [15] demonstrated that it is possible to verify a nontrivially-sized microkernel. However, the kernel's certification is limited: e.g., the virtual memory manager is left uncertified. With recent advances in certified software [23, 10], it has become practical to develop a fully-certified OS kernel. CertiKOS is designed to leverage and showcase these certification techniques.

## 3. The Kernel Design

Figure 1 illustrates the architecture of CertiKOS. The design currently handles only the most common types of resources in cloud environments: CPU cores for computation, RAM and disk storage, and network controllers for communication. CertiKOS implements a low-level layer of abstraction over physical resources. Both the provider's resource management facilities and cloud customers' software execute atop this kernel, which protects guests from each other and from the provider's management software. The rest of this section highlights relevant design details and rationale.

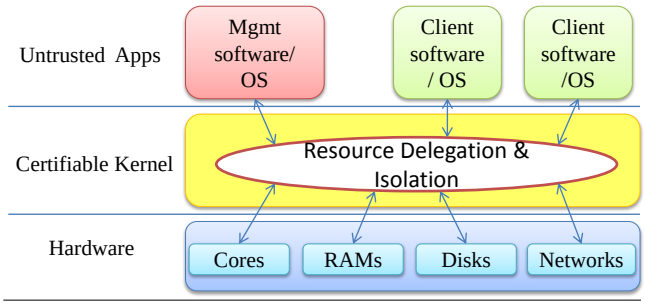


**Figure 1.** Overview of CertiKOS

### 3.1 Trust Model

We leave physical attacks out of scope, assuming that hardware is trusted and physically well-protected, a reasonable assumption in data centers designed for security. End users establish trust in CertiKOS via two mechanisms: by verifying the static correctness of the CertiKOS kernel, and by verifying its correct deployment at runtime. We will apply recent advances in certified software to produce a formal, machine-checkable proof of the correctness of the kernel's resource delegation and isolation properties. By distributing these formal proofs with CertiKOS, end users can confirm its security without trusting the kernel developer. The user relies on a standard hardware-based root of trust, such as a TPM [24], to attest that the cloud hardware is running the formally certified kernel. We assume that the provider's management software and all guest software are untrusted, and may be compromised or malicious.

### 3.2 Design Goals and Rationale

The main design principle underlying CertiKOS is to separate high-level resource management software from low-level resource delegation and protection mechanisms. CertiKOS separates the traditional resource management mechanisms implemented by existing hypervisors, such as Xen's Dom0, into two trust levels: a certified low-level layer implementing only minimal resource delegation and protection mechanisms, and a higher-level resource management layer, which is trusted in conventional hypervisors but untrusted in CertiKOS. The certified kernel's sole purpose is to implement the resource delegation policies specified by the untrusted, server-provided resource management software, while protecting the computation integrity and information security of cloud guests from each other and from the provider's management software.

Since management and guest software are equally untrusted by the kernel, the kernel implements isolation mechanisms ensuring that each untrusted domain has access only to its own resources. To implement this isolation, CertiKOS maintains ownership records for all hardware resources. When an end user requests cloud resources, the provider's management software allocates the resources requested by the client, then uses CertiKOS's resource management interface to delegate the resources to the client's execution. This resource management interface, accessible only by the provider-controlled management software, allows the provider to update the CertiKOS's resource ownership records by reassigning ownership of specific resources to the client. For example, when the management software starts a hosted client, it might use the CertiKOS management interface to assign a particular set of CPU cores, a particular physical memory range, a particular disk partition, and a particular network interface controller (NIC) to the client. This management interface is designed—and certified—to prevent information leaks. The provider's management software can later revoke memory or disk storage it assigned the client, for example, but the kernel

guarantees that the reassigned storage is appropriately "scrubbed" before being returned to provider control.

### 3.3 Resource Abstraction and Delegation

Cloud users primarily require three types of resources: computation, storage (RAM and disk), and networking. CertiKOS provides interfaces to the provider's resource management layer to assign and delegate access to these resources, but once assigned, the kernel exposes the assigned resources directly to the current owner, with minimal virtualization or abstraction in the kernel.

The CertiKOS kernel implements no dynamic resource scheduling, such as multiplexing a single CPU core among multiple guests, as current VMs often do. Instead, it relies on the provider's untrusted resource management code to perform any necessary resource allocation and scheduling at relatively coarse granularity. This design principle relies on the fact that modern server hardware usually provides the relevant resources—memory, CPU cores, and real or hardware-virtualized NICs—in sufficient number to make spatial partitioning feasible. We expect CertiKOS's focus on spatial rather than temporal partitioning also to increase the architecture's resistance to side-channel attacks in the cloud [19].

**CPU Cores:** CertiKOS allocates CPU resources spatially, at core granularity. In the current design, CertiKOS simply assigns one guest to each core, though future extensions could allocate CPUs more flexibly. CertiKOS provides interfaces for management software to allocate CPU cores to clients and revoke them.

**RAM and Disks:** CertiKOS abstracts both RAM and disks as memory space for untrusted management software and guests. The kernel includes no file system, leaving this functionality to untrusted guest software. CertiKOS exposes only interfaces supporting delegation of and protected access to memory and disk storage.

**Networking:** CertiKOS exposes interfaces allowing a provider's management software to give clients access to the provider's shared network infrastructure. The current design assumes that server hardware provides either enough physical NICs, and/or hardware-virtualized NICs, to dedicate at least one NIC to each client without software multiplexing. CertiKOS may be enhanced in the future to provide its own dynamic multiplexing of network interfaces, but this is not a high priority since hardware-based NIC virtualization is becoming increasingly common and inexpensive.

### 3.4 Resource Isolation

The provider's management software uses the trusted kernel's delegation interfaces to allocate and revoke resources according to the provider's policy. The trusted kernel in turn updates its ownership records to restrict each untrusted domain's resource access appropriately using standard protection and virtualization techniques. At runtime, CertiKOS uses its ownership records to check the permission on all explicit access requests, and to configure hardware-based protection mechanisms such as MMU and IOMMU hardware. In this way CertiKOS enforces resource isolation among applications and prevents information leakage.

Figure 2 illustrates the conceptual isolation model. A client's resources must be allocated in advance (step 1), otherwise the kernel denies the client's access request at runtime (step 5). When the client attempts to access its own resource, it invokes the kernel (step 2), which intercepts the request and checks the appropriate ownership record (step 3). If the client is the current owner of the requested resource, the kernel services the access request (step 4).

The kernel imposes access permission checks even on access requests by the provider's management software (step 6). Although

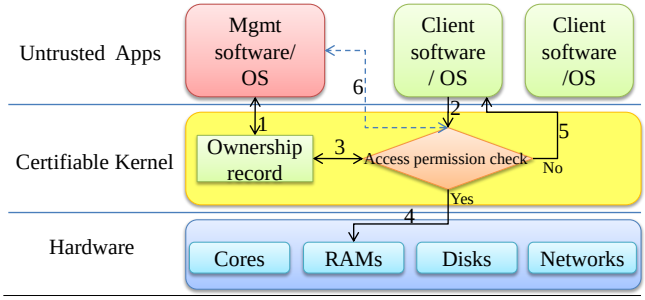


**Figure 2.** Permission check for access requests
1. Resource allocation/revocation; 2. Access request from a client; 3. Query the ownership record; 4. Authorize the request; 5. Deny the unauthorized request and return a fault message; 6. Access request from the management software.

the management software can allocate and revoke cloud hardware resources, it is not allowed to access resources owned by clients.

CertiKOS intercepts access requests based on resource types. In the current version of CertiKOS, CPU cores are handled by the kernel's minimal CPU multiplexer, and are allocated at a granularity of entire cores. Clients may choose to implement their own finer-grained scheduling atop CPU cores they own.

CertiKOS employs memory management hardware to partition and isolate guests' access to physical memory. In the current version, CertiKOS uses the AMD Secure Virtual Machine [3] facilities, setting up hardware page tables to represent memory ownership and relying on the hardware to translate guest virtual addresses to host physical addresses. Guests manage and access their assigned physical memory in the same way as on a physical platform, enabling clients to run traditional guest operating systems. CertiKOS uses IOMMU (Input/Output Memory Management Unit) hardware to protect DMA-based access requests from guests via physical or hardware-virtualized I/O devices such as NICs.

### 3.5 Resource Management and Usage

CertiKOS isolates management and guest software into strongly isolated runtime environments. The kernel gives the provider's management software the special privileges required to allocate and revoke resources, in order to implement the provider's high-level business model. Clients in turn manage their own assigned resources using their own mechanisms, such as those in a conventional guest OS. The current CertiKOS design focuses on managing and protecting a single multicore machine, although we expect to extend it in the future to offer secure clustering across data center networks.

A client may request and be assigned resources in several ways:

- Explicitly, using IPC to contact the management software and requesting resources. It is up to the client and the management to agree on the structure of the message. The kernel is not involved, and does not offer privacy guarantees to inter-domain messages. A client might request additional memory this way, for example.
- Explicitly, by including the resources in its initial loading instructions. When the management software loads the client, the client's object format (which is up to the provider and the client to agree on) may specify resources that the process requires. The management then checks that the specified resources observe its policy, and instructs the kernel to allow the client access to them.
- Implicitly. The client program may try to access a resource, initially causing a fault caught by the kernel. The kernel notifies the management program of the fault event, scrubbed of any

private client data. The management interprets the fault, assigns the requested resources to the client if appropriate by asking the kernel to update the ownership records, then restarts the client with the additional resources.

***Resource Revocation*** Given unrestricted ability to revoke arbitrary subsets of a client's resources at any time, compromised management software might attempt to interfere with a client's correct execution, and/or attempt to learn information about a guest through side-channel attacks. Management software might selectively revoke assigned resources at a provider-chosen moment in time, for example, and monitor how this revocation affects visible guest behavior in an attempt to "probe" what a guest is doing based on its resource usage.

To guard against such side-channel attacks, CertiKOS gives untrusted management software only one *non-consensual* means of revoking guest resources: by destroying the entire guest and revoking all its resources at once, in "all-or-nothing" fashion. Such an event essentially represents a provider-induced virtual node failure, which the kernel guarantees will operate in a fail-stop fashion without leaking any information about the former guest's operation.

If the provider wishes to revoke resources from guests at finer granularity, it can communicate explicitly with its guests to *request* that they give up certain resources gracefully. The guest saves any state the resources may have held (for memory or disk), and notifies the kernel when it is ready to relinquish the resources, which in turn passes resource ownership back to the provider. If the guest fails to respond in a reasonable time, according to a policy agreed between provider and client, the provider can always escalate to an "all-or-nothing" revocation without the guest's consent.

## 4. Certifying the Kernel

To verify CertiKOS, we will take advantage of its minimalistic design and several recent advances in software certification [23].

First, we will not certify what we do not have to. Much of the complexity in cloud virtualization lies in resource management algorithms. CertiKOS delegates this functionality to the untrusted management layer, implementing only permission checking in the kernel. Thus, we can omit sophisticated policy management and scheduling tasks from the kernel without sacrificing functionality, leaving less kernel code to certify formally.

Second, we are designing the kernel ground-up for certification, instead of trying to certify existing software. Instead of disentangling complex existing kernels, therefore, we are developing a kernel specifically to be easy to reason about. In particular, the kernel is designed to be highly modular, with strong and readily formalizable invariants defining module boundaries. Although conventional wisdom may suggest that such strong modularity risks incurring high performance overhead, our formal layering and modular reasoning techniques are capable of expressing modules whose formal boundaries do not actually impose additional code at each layer or interface, which may mitigate this performance risk. Further, even if the resulting certifying kernel achieves far from optimal performance, CertiKOS's focus on leveraging hardware protection and virtualization mechanisms, and spacial rather than temporal resource assignment, leaves the certified kernel responsible mostly for occasional setup, teardown, and guest fault handling, with hardware handling most of the performance-critical resource access paths.

We will employ domain-specific logics to certify different CertiKOS modules through the use of OCAP [10], a framework designed for this purpose. This approach allows us to integrate several frameworks that have proven effective for certifying programs involving dynamically allocated memory [27], interrupts [11], non-interfering concurrent threads [11], and self-modifying code [5].

Another technique we will employ allows certified linking of components verified at different abstraction levels. Since only the bootloader loads code dynamically, for example, we need use a self-modifying code framework only for bootloader verification. The rest of the kernel can use a fixed code heap abstraction. We can then link these modules together across their different levels of abstraction, such that the entire resulting code is certified.

We are developing an approach for merging Decentralized Information Flow Control (DIFC) [17] with ideas from Separation Logic [18]. We will develop a logic in which we can specify and prove the correctness of information flow policies. We will apply this logic to CertiKOS to prove that the kernel properly enforces isolation. This guarantees that our kernel cannot, for example, accidentally leak data from one client's memory to another's.

We will use a Flume-like label model [16] to track the origin of all data. In this model, memory locations are "tainted" with tags representing clients. If the kernel reads data from client A, that data is tainted with tag A. If the kernel then attempts to write this data into client B's memory, which is tainted with tag B, our logic catches this bug, causing certification to fail. Of course, there are situations in which data must be passed across domains, such as IPC between client and provider software. As in any useful DIFC system, our logic will allow controlled *declassification* of data in appropriate situations to handle such cases explicitly. Our logic will make all declassification explicit, allowing for formal certification of high-level declassification policies. For example, we might prove that the kernel never leaks data from one client to another, *except for* IPC messages. It then becomes the client's responsibility to avoid leaking information via IPC. For more discussion of the role and necessity of declassification in DIFC, see [20].

## 5. Implementation

The CertiKOS design presented here is just a first step in a larger project to develop a fully-certified, practical OS kernel. As the first stage of this project, we wished to implement a lightweight and modular kernel with minimal functionality to facilitate certification. Our minimal kernel design reflects the goal of separating out and surgically confronting critical cloud security problems.

We are now building a prototype based on PIOS[2], by further modularizing the kernel and minimizing global interdependencies. Figure 3 illustrates this conceptual simplification by comparing the dependency graphs of the old and new kernels.

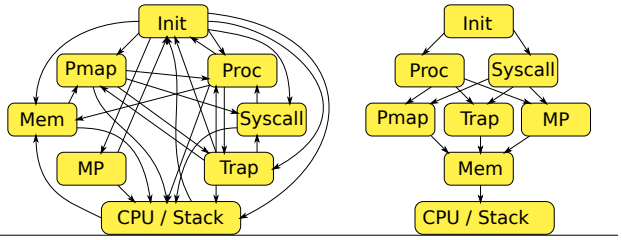


**Figure 3.** Invariant dependencies of PIOS vs. our prototype

We believe that a massive redesign of core code of the kernel will be crucial for verification and reasoning about isolation properties.

Our prototype allows us to manage the client's use of CPU cores, limit its use of memory, and guarantee that management and client code cannot access each other's memory. Continuing development is leading in two main directions. First, we will improve and generalize the features of our kernel. Our permission system is currently

ad hoc, and does not readily extend to new resources. Second, we will more rigorously separate the kernel components, facilitating the formal specification and verification of each component.

Atop the kernel, we are building an untrusted management application as an example. This application serves as a demo showing how a cloud provider might use our kernel. Our aim for this demo is to illustrate the kernel's ability to allow the management application to safely run the clients' applications, and manage their access to resources, while formally guaranteeing security and isolation.

## 6. Conclusion

This paper presents CertiKOS, a certified kernel for secure cloud computing. CertiKOS handles resource delegation and isolation, and protects clients from each other and from the provider. The current version targets a single platform; we will later investigate cross-machine clustering and migration.

***Acknowledgment*** We thank anonymous referees for their suggestions and comments on an earlier version of this paper. This research is based on work supported in part by the DARPA CRASH grant FA8750-10-2-0254 and NSF grants CNS-1017206, CNS-0915888, CNS-0910670, and CCF-0811665. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

## References

[1] Intel virtualization technology (VT). `http://www.intel.com/technology/virtualization/technology.htm`.

[2] Parallel instructional operating system. `http://zoo.cs.yale.edu/classes/cs422/pios`.

[3] AMD. AMD64 virtualization codenamed "Pacifica" technology — secure virtual machine architecture reference manual. Tech. Rep. Publication Number 33047, Revision 3.01, AMD, May 2005.

[4] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *17th ACM Conference on Computer and Communications Security, 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., pp. 38–49.

[5] CAI, H., SHAO, Z., AND VAYNBERG, A. Certified self-modifying code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pp. 66–77.

[6] CIRCLEID. Survey: Cloud Computing No Hype, But Fear of Security and Control Slowing Adoption. `http://www.circleid.com/posts/20090226_cloud_computing_hype_security/`.

[7] CVE. CVE-2008-2100: VMware Buffer Overflows in VIX API Let Local Users Execute Arbitrary Code in Host OS. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2100`.

[8] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'07)* (Brighton, UK, Oct. 2005), ACM, pp. 17–30.

[9] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pp. 251–266.

[10] FENG, X. *An Open Framework for Certified System Software*. Ph.D. thesis, Department of Computer Science, Yale University, 2007.

[11] FENG, X., SHAO, Z., DONG, Y., AND GUO, Y. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pp. 170–182.

[12] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pp. 193–206.

[13] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, Redmond, WA, USA, Oct. 2005.

[14] KELLER, E., SZEFER, J., REXFORD, J., AND LEE, R. B. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proc. 37th International Symposium on Computer Architecture (37th ISCA'10)* (Saint-Malo, France, June 2010), ACM SIGARCH, pp. 350–361.

[15] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)* (Big Sky, MT, Oct. 2009), ACM SIGOPS, pp. 207–220.

[16] KROHN, M. N., YIP, A., BRODSKY, M. Z., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard os abstractions. In *SOSP* (2007), pp. 321–334.

[17] MYERS, A. C., AND LISKOV, B. A decentralized model for information flow control. In *SOSP* (1997), pp. 129–142.

[18] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *LICS* (2002), pp. 55–74.

[19] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pp. 199–212.

[20] SABELFELD, A., AND SANDS, D. Declassification: Dimensions and principles. *Journal of Computer Security 17*, 5 (2009), 517–548.

[21] SAILER, R., JAEGER, T., VALDEZ, E., CACERES, R., PEREZ, R., BERGER, S., GRIFFIN, J. L., AND DOORN, L. V. Building a mac-based security architecture for the xen open-source hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference* (2005), pp. 276–285.

[22] SANTOS, N., GUMMADI, K. P., AND RODRIGUES, R. Towards trusted cloud computing. In *Proceedings of the conference on Hot topics in cloud computing*, HotCloud'09, pp. 3–3.

[23] SHAO, Z. Certified software. *Commun. ACM 53* (December 2010), 56–66.

[24] TRUSTED COMPUTING GROUP. TPM main specification. `http://www.trustedcomputinggroup.org`, Feb. 2005.

[25] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), SP '10, pp. 380–395.

[26] WOJTCZUK, R. Subverting the Xen hypervisor. In *Blackhat* (2008). `http://www.invisiblethingslab.com/resources/bh08/part1.pdf`.

[27] YU, D., HAMID, N. A., AND SHAO, Z. Building certified libraries for pcc: dynamic storage allocation. *Sci. Comput. Program. 50* (March 2004), 101–127.

[28] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation* (Nov. 2006).

# An Efficient Software Shared Virtual Memory for the Single-chip Cloud Computer*

Junghyun Kim, Sangmin Seo, and Jaejin Lee
School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
{junghyun, sangmin}@aces.snu.ac.kr, jlee@cse.snu.ac.kr

## ABSTRACT

The Single-chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. The SCC is based on a message passing architecture and does not provide any hardware cache coherence mechanism. Software or programmers should take care of coherence and consistency of a shared region between different cores. In this paper, we propose an efficient software shared virtual memory (SVM) for the SCC as an alternative to the cache coherence mechanism and report some preliminary results. Our software SVM is based on the commit-reconcile and fence (CRF) memory model and does not require a complicated SVM protocol between cores. We evaluate the effectiveness of our approach by comparing the software SVM with a cache-coherent NUMA machine using three synthetic micro-benchmark applications and five applications from SPLASH-2. Evaluation result indicates that our approach is promising.

## 1. INTRODUCTION

As the number of cores increases in a chip multiprocessor, the on-chip interconnect becomes a major performance and power bottleneck. It takes up a significant amount of the total power budget. A complicated hardware cache coherence protocol worsens the situation by increasing the amount of messages that go through the interconnect. For this reason, there have been many studies on the relationship between on-chip interconnects and cache coherence protocols for manycores. But, most of them increase the chip design complexity and result in introducing high hardware validation costs. Some recent manycores, such as the Intel Single-chip Cloud Computer (SCC)[4], choose a message passing architecture that does not require hardware cache coherence mechanism although this sacrifices ease of programming.

The SCC experimental processor[4] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. Figure 1 describes the overall organization of the SCC. It has 24 core tiles arranged in a 6x4 array and four DDR3 memory controllers. Each tile consists of two P54C IA cores and 16KB message passing buffer (MPB). Each core has 16KB L1 instruction and data caches and a 256KB L2 cache. MPB is designed for fast communication between cores. Intel provides a customized Linux running on a core and an MPI-style communication library called RCCE[14].

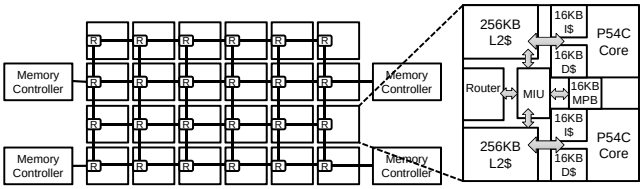


**Figure 1: The organization of the SCC.**

Each memory controller supports up to 16GB DDR3 memory. While this implies that a maximum capacity of 64GB memory is provided by the SCC, each core is able to access only 4GB of memory because it is based on the IA-32 architecture. To overcome this limitation, the SCC allows each core to alter its own memory map. Each core has a lookup table, called LUT, which is a set of configuration registers that map the core's 32-bit physical addresses to the 64GB system memory. Each LUT has 256 entries, and each entry handles a 16MB segment of the core's 4GB physical address space. An LUT entry can point to a 16MB system memory segment, an MPB, or a memory-mapped configuration register. Each core's LUT is initialized during the bootstrap process, but the memory map can be altered by any core at any time. Consequently, the software can freely map any memory region as a shared or private region. The SCC does not provide any cache coherence mechanism. This implies that software or programmers should take care of coherence and consistency of a shared region between different cores.

A software shared virtual memory (SVM) system[1, 5, 7, 8, 9, 10, 11, 13] can be one of the alternatives to the hardware cache coherence mechanism. Software SVMs were originally developed for distributed memory multiprocessors to provide an illusion of a single, globally shared address space for the user. In this paper, we propose an efficient software SVM for the SCC and report some preliminary results. Our software SVM exploits the commit-reconcile and fence (CRF) memory model[12] to guarantee release consistency[3]. Optimizing compiler techniques make our software SVM easy to use. Since the memory consistency model is implemented by software, any memory consistency model can be implemented in our SVM without any hardware sup-

*This work was supported by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea funded by the Korean government (Ministry of Education, Science and Technology). ICT at Seoul National University provided research facilities for this study.


APSys 2011, July 11-12, 2011, Shanghai, China

port if it can be specified by the CRF memory model. We evaluate the effectiveness of our approach by comparing the software SVM with a cache-coherent NUMA machine using three synthetic micro-benchmark applications and five applications from SPLASH-2[15].

## 2. RELATED WORK

There have been many studies done on software SVM systems for distributed memory multiprocessors[1, 5, 7, 8, 9, 10, 11, 13]. Most of them guarantee memory consistency and coherence at the page level. They are based on a page fault handling mechanism to detect pages that are accessed by a processor. They also support multiple writers protocols to avoid page pingponging between processors due to false sharing. These protocols are typically based on maintaining a copy of the original page (e.g., creating a twin) and comparing the modified page to the copy (e.g., making a diff). There are two major differences between our proposal and theirs. First, we do not maintain twins and we do not have any process for making diffs. Based on the CRF memory model, a compiler or programmer identifies the data that should be updated to the main memory or that should be brought from the main memory. Second, our proposal does not require a complicated protocol between processors because the main memory can be configured to be physically shared between different SCC cores by modifying the LUT (however, note that the cores in the SCC do not have a single, globally shared address space). What we need to do is just copying the data between a private memory region and the shared memory region in the SCC according to a much simpler protocol.

Inagaki *et al.* [6] propose a compiler technique to improve the performance of a page-based software SVM system. The compiler generates *write commitments* that are sent to all other nodes to immediately update or invalidate the modified memory region when a synchronization operation is performed. Similar to their approach, our approach is based on a compiler technique. However, thanks to the reconcile operation before a read operation in our approach, the updates do not need to be immediately reflected to other local pages at a synchronization point. Moreover, our approach does not need to update or invalidate the entire page.

## 3. MEMORY CONSISTENCY

In this section, we briefly describe the technique that is used to guarantee memory consistency and coherence in our software SVM.

The CRF memory model is a mechanism-oriented memory model[12]. In addition to a semantic cache, called a *sache*, it has three consistency operations: commit, reconcile, and fence. All load and store operations are executed only on saches locally. These operations cannot access the main memory directly. To update the main memory, programmers or compilers should insert commit or reconcile operations appropriately. The model assumes memory accesses can be reordered as long as data dependences are preserved.

A commit operation updates the local data to the main memory if the address is cached in the sache and its state is dirty. A reconcile operation purges the data in the sache if the address is cached and its state is clean. Fence operations are used to enforce ordering between memory accesses. These commit, reconcile, and fence operations are realized in system calls in our software SVM. Our software SVM defines release consistency (RC)[3] for its memory consistency model.

```
for(i = start; i < end; ++i) {
    a[i] = a[i] + b[i];
    priva += a[i];
}

lock(s);
for(i = 0; i < n; ++i) {
    c[i] += priva;
}
unlock(s);
```

(a)

```
shared(&a, &b, &c);

for(i = start; i < end; ++i) {
    reconcile(&a[i],1,0,sizeof(a[i]));
    reconcile(&b[i],1,0,sizeof(b[i]));
    a[i] = a[i] + b[i];
    priva += a[i];
    commit(&a[i],1,0,sizeof(a[i]));
}

lock(s);
for(i = 0; i < n; ++i) {
    reconcile(&c[i],1,0,sizeof(c[i]));
    c[i] += priva;
    commit(&c[i],1,0,sizeof(c[i]));
}
unlock(s);
```

(b)

```
shared(&a, &b, &c);

reconcile(&a[start],1,0,sizeof(a[start])*(end-start));
reconcile(&b[start],1,0,sizeof(b[start])*(end-start));
for(i = start; i < end; ++i) {
    a[i] = a[i] + b[i];
    priva += a[i];
}

lock(s);
reconcile(&c[0],1,0, sizeof(c[0])*n);
for(i = 0; i < n; ++i) {
    c[i] += priva;
}
commit(&a[start],1,0,sizeof(a[start])*(end-start));
commit(&c[0],1,0,sizeof(c[0])*n);
unlock(s);
```

(c)

**Figure 2: Code translation based on the CRF model to guarantee RC. (a) Original code. (b) After inserting commits and reconciles. (c) The final code.**

To implement RC, the CRF model defines acquire and release operations in RC as follows:

```
release(s)  ≡  commit(*);preFenceW(s);unlock(s);
acquire(s)  ≡  lock(s);postFenceR(s);reconcile(*);
```

To define a release operation in RC, all dirty data in the sache must be updated to the main memory by `commit(*)` before `unlock(s)` is performed. `preFenceW(s)` makes any memory access preceding it to be completed before writing to location `s`. To define an acquire operation in RC, after `lock(s)` has been performed, `postFenceR(s)` makes all load operations to location `s` to be completed before any memory access following it is performed. Then, `reconcile(*)` purges all the clean data in the sache. In our software SVM, `preFenceW(s)` and `postFenceR(s)` are combined with `unlock(s)` and `lock(s)`, respectively. That is, `unlock(s)` and `lock(s)` perform these fence operations in addition to their original function.

We add commit and reconcile system calls to the SCC Linux kernel. When a commit or reconcile operation occurs in the application, the request is sent to the kernel with the following arguments:

- Start address: the address of the first data item to commit or reconcile.

- Number of data items: how many data items to commit or reconcile.
- Stride: the distance between two consecutive data items in bytes.
- Size: the size of a data item in bytes to commit or reconcile.

Consider the code in Figure 2 (a) and assume arrays `a`, `b`, and `c` are shared. To run the code on our software SVM, the compiler or programmer declares the arrays as shared arrays (Figure 2 (b)). Then, a reconcile operation is inserted before every read reference to the shared arrays, and a commit operation is inserted after every write reference to the shared arrays. As an optimization, reconcile and commit operations can be reordered and coalesced according to the rules specified in the CRF model. Since we assume RC, commit operations are moved as close as possible to `unlock(s)` and reconcile operations are moved as close as possible to `lock(s)` (Figure 2 (c)). Then, those operations are coalesced together. Moving commit and reconcile operations as close as possible to synchronization operations increases the chance of coalescing those operations. This significantly reduces the system call overhead.

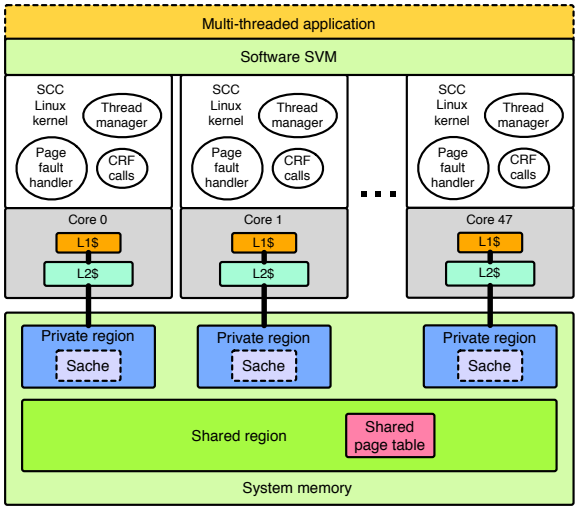


**Figure 3: The organization of the runtime system.**

## 4. RUNTIME SYSTEM

Figure 3 shows the organization of our runtime. SCC Linux is running on each core. The system memory space allocated to each core is divided into two regions: private and shared. The private region is private to the core, and a space in the private region is dedicated to the Linux OS. The remaining space in the private region is allocated to a sache. When a multi-threaded application is running, its shared data is placed in the shared region. The shared region is configured to be uncacheable to L1 and L2 caches by the OS. We modify the SCC linux kernel and add some features (e.g., CRF system calls) to support executing multi-threaded applications. The kernel on each core maintains two page tables: a private page table (PPT) and a shared page table (SPT). The PPT is private to each core and is similar to the page table found in a conventional virtual memory system. It maps each core's virtual address to a physical address in the sache. The SPT is shared between

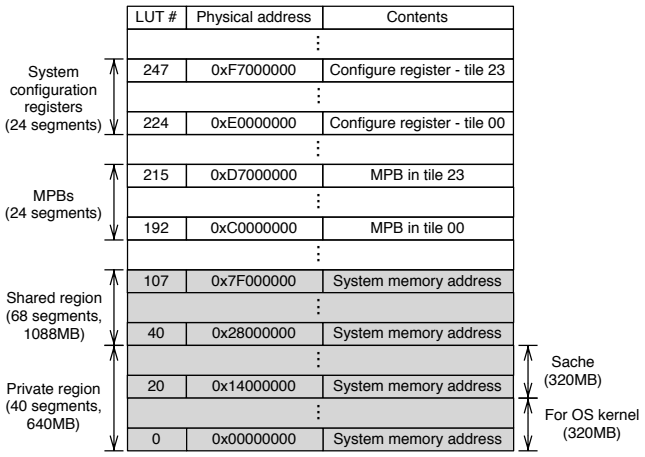


**Figure 4: The LUT for an SCC core.**

all cores and maps each core's virtual address to a physical address in the shared region. A space in the shared region is allocated to the SPT itself.

**LUT setting.** Figure 4 shows the LUT used by each SCC core in our software SVM. The gray area in the LUT shows the mapping between each core's physical address space to the system memory address space for the private and shared regions. As a result of the LUT setting, each core sees the same system memory segment with the same LUT index in the shared region and a different system memory segment with the same LUT index in the private region. The set of system memory segments allocated to the private region of a core is disjoint from that of another core.

**Thread manager.** When we run a multi-threaded application on top of our software SVM, we make each core run the same copy of the application. When the system function `exec` is invoked, the kernel checks whether an environment variable `SCC_SVM` is set. If so, this indicates that the application will be running on our software SVM. Then, only core 0 (say, master) continues its execution. Other cores are waiting for a wake-up event from the master. When the master creates a thread (e.g., by calling `pthread_create`), it sends a wake-up message to a waiting core in a round-robin manner. The message contains information about the address of the function to be executed, arguments of the function, and the stack pointer, to run the created thread on the target core. Then, the woken-up core starts executing the designated function. This process is managed by the thread manager added to the SCC Linux kernel. The thread manager provides POSIX thread library routines.

**Page fault handler.** We also modify the page-fault handler in the SCC Linux kernel. To explain the function of the modified page-fault handler, consider the scenario shown in Figure 5. Assume that core i is the first core that accesses a shared page $p$ with the virtual address $VA_p$ among all cores. In turn, core j accesses $p$ for its first time with $VA_p$. Since the software SVM provides a single, shared address space between all the cores, each core has the same virtual address for the same shared page.

When core i accesses the shared page $p$, a page fault occurs. The first part of the page fault handling process is similar to that of the conventional process and allocates a page frame $p_i$ in its sache (❶). Then, the handler initializes $p_i$ (e.g., $p$ is a page in .bss section). Following this, the handler looks up the SPT with $VA_p$ after obtaining the page lock for the SPT entry of $VA_p$. The SPT entry for $VA_p$ is

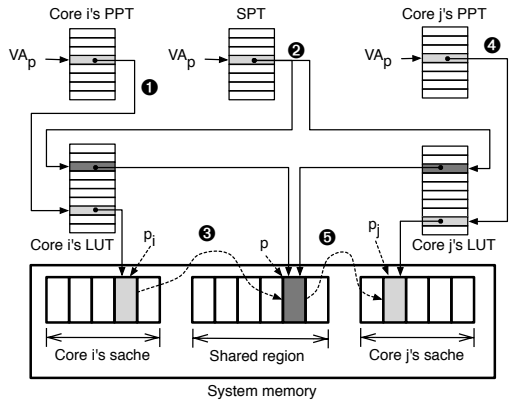


**Figure 5: Page allocation.**

invalid because $p$ is accessed for the first time. The handler sets up the SPT entry to point to the LUT index of $p$ (❷). Then, it copies $p_i$ to $p$ (❸) and releases the page lock. Core i starts to access the private copy $p_i$ of $p$ in its sache.

Now, the page-fault handler of core j acquires the page lock. Since the SPT entry of $VA_p$ already points to the LUT index that is associated with $p$, after allocating the page frame $p_j$ in core j's sache (❹), the handler copies $p$ to $p_j$ (❺). Then, core j starts to access the private copy $p_j$ of $p$ in its sache.

**Table 1: Applications Used**

| Code | Problem Size |
|---|---|
| FFT | 4M points |
| LU/C | 4096x4096 matrix, 16x16 blocks |
| LU/N | 4096x4096 matrix, 16x16 blocks |
| RADIX | 16M integers, radix 1024 |
| OCEAN/C | 1026x1026 ocean |

**Table 2: System Configurations**

| Cache-coherent NUMA Machine (cc-NUMA) | |
|---|---|
| CPU | AMD Opteron 6174 |
| Core frequency | 2.2GHz |
| # of sockets | 4 |
| Cores | 12 per Socket, total 48 cores |
| L1 I-cache | 64KB per core |
| L1 D-cache | 64KB per core |
| L2 Cache | 512KB per core |
| L3 Cache | 12MB per socket |
| Main memory | 96GB |
| Compiler | GCC 4.1.2 |
| **Single-Chip Cloud Computer (SCC)** | |
| Core type | P54C IA core |
| Core frequency | 533Mhz |
| # of Cores | 48 |
| L1 I-cache | 16KB |
| L1 D-ache | 16KB |
| L2 Cache | 256KB |
| Main memory | 32GB |
| Compiler | GCC 3.4.5 |

# 5. EVALUATION

In this section, we evaluate our software SVM by comparing its performance with a cache-coherent NUMA machine (cc-NUMA).

## 5.1 Methodology

**Machines.** The architecture configurations of the SCC and the cc-NUMA machine are described in Table 2. The cc-NUMA machine has four 12-core AMD Opteron processors in a single system. It is supported by a broadcast-based hardware cache coherence protocol[2].

**Benchmark applications.** In addition to three synthetic micro-benchmark applications: FalseSharing, NoFalseSharing, and LocalDataAccess, we use five applications from the SPLASH-2 benchmark suite[15] for the evaluation. The problem size of each SPLASH-2 application is shown in Table 1. FalseSharing is a program that increments each element of an array in a loop with many iterations. The size of an array element is a single byte. The array is equally divided into $c \cdot N$ chunks, where $N$ is the number of threads and $c$ is an integer greater than or equal to 1. Each thread is in charge of $c$ chunks and the chunks are distributed cyclicly to each thread. The chunk size is smaller than the cc-NUMA's L2 cache line size (64 bytes). Thus, a significant amount of false sharing occurs in FalseSharing. NoFalseSharing is a program that is similar to FalseSharing, but the chunk size is exactly the same as the cc-NUMA's L2 cache line size and each cache block is aligned to a cache line boundary. LocalDataAccess is a program that does not cause any L1 cache misses but cold misses. Its data access always hits in the L1 cache. We would like to observe maximal scalability with LocalDataAccess.

**Software SVM for the SCC.** We implement the runtime for the software SVM by modifying the SCC Linux kernel. We insert commit and reconcile operations manually into the applications and optimize them by hand according to the rules specified in the CRF memory model.

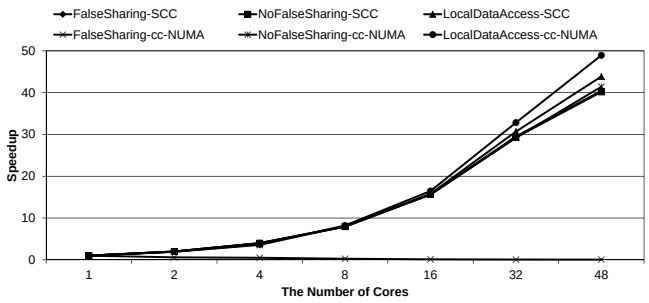


**Figure 6: Speedup of the micro-benchmark applications.**

## 5.2 Result

Figure 6 shows the evaluation result with the three micro-benchmark applications. The speedup is obtained over a single core. For FalseSharing, our software SVM does not suffer from false sharing as the number of cores increases while cc-NUMA suffers from heavy false sharing. Even though pages are falsely shared in our software SVM for FalseSharing and NoFalseSharing, after an SCC core obtains a page in its sache, all accesses to the page occur in the sache, and the updates are committed altogether later to the system memory without any diff process. Interestingly, false sharing is not a source of the scalability bottleneck in our approach.

When we see the speedup of LocalDataAccess, both of the SCC and cc-NUMA scale almost linearly. When the number of cores increases (e.g., 32 and 48), the speedup of the SCC is a little bit worse than that of the cc-NUMA. This is due to the overhead of looking up the SPT (e.g., page lock contention and uncached system memory access). In addition, the overhead of copying pages from the system memory to the sache (e.g., uncached system memory access) takes more portion in the total execution time as the number of cores increases.

Figure 7 shows the evaluation result with the five SPLASH-2 benchmark applications. FFT, RADIX, and OCEAN/C

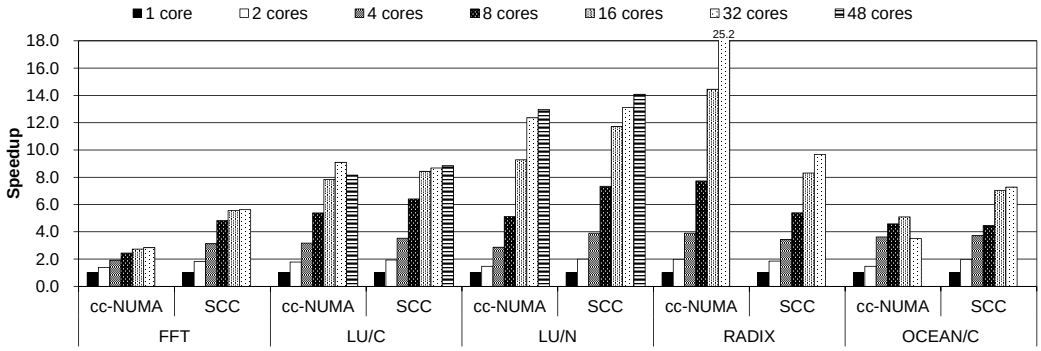


**Figure 7: Speedup of the SPLASH-2 benchmark applications.**

do not have the result for 48 cores because these applications require the number of cores to be a power of two. For all applications but RADIX, the SCC with our software SVM shows better scalability than the cc-NUMA machine. As the number of cores increases, the speedup also increases for the SCC. However, the speedup with 48 cores in cc-NUMA is often worse than that with 32 cores. This is due to false sharing and manifests in the speedup of LU/C and LU/N with 48 cores because LU/N's data accesses are optimized to avoid false sharing.

RADIX executes many spin-wait loops. A flag variable is read repeatedly in the spin-wait loop. In our software SVM, implementing a spin-wait loop requires a reconcile operation to be placed in the loop for the flag variable. Consequently, frequent updates to the flag variable in the sache through the reconcile system call incur a significant overhead. This can be solved with either hardware support or by modifying the application with a post-wait synchronization mechanism.

# 6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose an efficient software SVM for the SCC. It is based on the CRF memory model and guarantees release consistency. Legacy multi-threaded applications can be run on top of the software SVM without any modification. Due to the CRF memory model and compiler support, we do not need to implement a complicated multiple writer protocol that manages the process of creating twins and making diffs. This makes the runtime much simpler. Since the memory consistency model is implemented by software, any memory consistency can be implemented in our software SVM if it can be specified by the CRF model. Based on the preliminary evaluation result that compares our software SVM with a cc-NUMA machine, we foresee such a software SVM can be an alternative to a hardware cache coherence protocol for the SCC-like manycore architectures. As future work, we plan to develop a compiler and its techniques for optimizing commit and reconcile operations based on their properties. In addition, we plan to evaluate our software SVM with more multi-threaded applications.

# 7. REFERENCES

[1] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP '91: Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[2] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, 1990.

[4] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom1, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam2, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC' 10: Proceedings of the 57th International Solid-State Circuits Conference, February 2010*, 2010.

[5] L. Iftode and J. P. Singh. Shared virtual memory: progress and challenges. *Proceedings of the IEEE*, 87(3):498–507, March 1999.

[6] T. Inagaki, J. Niwa, T. Matsumoto, and K. Hiraki. Supporting software distributed shared memory with an optimizing compiler. In *Proceedings of the 1998 International Conference on Parallel Processing*, ICPP '98, 1998.

[7] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, January 1994.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA'92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[9] J. Lee, J. Lee, S. Seo, J. Kim, S. Kim, and Z. Sura. COMIC++: A Software SVM System for Heterogeneous Multicore Accelerator Clusters. In *HPCA'10: Proceedings of the 15th International Symposium on High Performance Computer Architecture*, January 2010.

[10] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: a coherent shared memory interface for cell be. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 303–314, 2008.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[12] X. Shen, Arvind, and L. Rudolph. Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 150–161, 1999.

[13] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[14] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, 1995.

# Linux kernel vulnerabilities: State-of-the-art defenses and open problems

Haogang Chen Yandong Mao Xi Wang Dong Zhou[†]
Nickolai Zeldovich M. Frans Kaashoek

MIT CSAIL [†]Tsinghua University

## ABSTRACT

Avoiding kernel vulnerabilities is critical to achieving security of many systems, because the kernel is often part of the trusted computing base. This paper evaluates the current state-of-the-art with respect to kernel protection techniques, by presenting two case studies of Linux kernel vulnerabilities. First, this paper presents data on 141 Linux kernel vulnerabilities discovered from January 2010 to March 2011, and second, this paper examines how well state-of-the-art techniques address these vulnerabilities. The main findings are that techniques often protect against certain exploits of a vulnerability but leave other exploits of the same vulnerability open, and that no effective techniques exist to handle semantic vulnerabilities—violations of high-level security invariants.

## 1. INTRODUCTION

An OS kernel is a part of the trusted computing base (TCB) of many systems. Vulnerabilities in the kernel itself can allow an adversary to bypass any kernel protection mechanisms, and compromise the system, such as gaining root access. Much research has gone into mitigating the effects of kernel vulnerabilities, but kernel vulnerabilities, and more importantly, kernel exploits, are still prevalent in Linux. This paper investigates where the research community may want to focus its attention, by analyzing past Linux kernel vulnerabilities, categorizing them, evaluating what defensive techniques might have been used to prevent them, and speculating on what the remaining open problems are.

Surprisingly few studies have been performed to understand the types of kernel vulnerabilities that occur in practice. A study by Arnold et al. [3] argues that every kernel bug should be treated as security-critical, and must be patched as soon as possible. Mokhov et al. explore how kernel programmers patch known vulnerabilities [19]. Christey and Martin report on vulnerability distributions in CVE [8]; our study is also based on CVE and our findings are consistent with that study, but ours focuses only on kernel vulnerabilities. Neither of the studies shed light on what techniques could be used to prevent unknown vulnerabilities from being exploited. In this paper, we present a case study of Linux kernel vulnerabilities discovered from January 2010 to March 2011. We categorize these vulnerabilities by the kind of programming mistake the developers made, and the impact it has on security.

Based on this list of kernel vulnerabilities, we perform a second case study, by examining how effective techniques proposed by researchers might be at mitigating vulnerabilities in the Linux kernel. These techniques include runtime mechanisms such as code integrity checks [22], software fault isolation [6, 15], and user-level device drivers [5], as well as bug-finding tools and static analysis [1, 2]. This examination is not empirical, but is based purely on our understanding of the techniques. Nonetheless, we believe it can be helpful in identifying what classes of vulnerabilities can already be solved, and what open problems remain.

The findings of our two studies are as follows. First, we find that there are 10 common classes of kernel vulnerabilities in Linux, which may lead to attacks ranging from arbitrary memory modifications to information leaks to denial-of-service attacks. Second, we find that about $2/3$ of the vulnerabilities are in kernel modules or drivers, and that conversely, $1/3$ of the bugs are found in the core kernel. Third, we find that no single existing technique can prevent all kernel vulnerabilities, and that a technique often mitigates some exploits of a vulnerability but does not address other exploits of the same vulnerability. Fourth, there are certain classes of vulnerabilities that are not addressed at all. For example, semantic vulnerabilities, where high-level security invariants are violated, are difficult to catch with state-of-the-art techniques that focus mostly on memory safety and code integrity.

## 2. LINUX KERNEL VULNERABILITIES

Figure 1 categorizes the 141 Linux kernel vulnerabilities published on the CVE list from January 2010 to March 2011 and the type of attacks that can exploit the vulnerability. Despite their diversity, most of these vulnerabilities fall into 10 categories, based on the kind of programming mistake the developers made, as listed in the first column. Since each vulnerability can often be exploited in several ways, we further categorize the exploits in several *attack* classes. *Memory corruption* typically allows an adversary to perform arbitrary operations in the kernel, and is thus a superset of other types of attacks, such as *policy violation*, *DoS*, *information disclosure*, and others. Therefore if a vulnerability leads to a memory-corruption attacks, we do not count it under other attack classes. The rest of this section discusses the types of vulnerabilities we have found.

**Missing pointer checks.** The kernel omits `access_ok` checks or misuses "faster" operations such as `__get_user`, which does not validate the value of user-provided pointers or index variables to ensure that they point to user-space memory only. These bugs enable unprivileged processes to read from or write to arbitrary kernel memory locations, leading to memory corruption (CVE-2010-4258),

| Vulnerability | Mem. corruption | Policy violation | DoS | Info. disclosure | Misc. |
|---|---|---|---|---|---|
| Missing pointer check | 6 | 0 | 1 | 2 | 0 |
| Missing permission check | 0 | 15 | 3 | 0 | 1 |
| Buffer overflow | 13 | 1 | 1 | 2 | 0 |
| Integer overflow | 12 | 0 | 5 | 3 | 0 |
| Uninitialized data | 0 | 0 | 1 | 28 | 0 |
| Null dereference | 0 | 0 | 20 | 0 | 0 |
| Divide by zero | 0 | 0 | 4 | 0 | 0 |
| Infinite loop | 0 | 0 | 3 | 0 | 0 |
| Data race / deadlock | 1 | 0 | 7 | 0 | 0 |
| Memory mismanagement | 0 | 0 | 10 | 0 | 0 |
| Miscellaneous | 0 | 0 | 5 | 2 | 1 |
| Total | 32 | 16 | 60 | 37 | 2 |

**Figure 1: Vulnerabilities (rows) vs. possible exploits (columns). Some vulnerabilities allow for more than one kind of exploit, but vulnerabilities that lead to memory corruption are not counted under other exploits.**

| Vulnerability | Total | core | `drivers` | `net` | `fs` | `sound` |
|---|---|---|---|---|---|---|
| Missing pointer check | 8 | 4 | 3 | 1 | 0 | 0 |
| Missing permission check | 17 | 3 | 1 | 2 | 11 | 0 |
| Buffer overflow | 15 | 3 | 1 | 5 | 4 | 2 |
| Integer overflow | 19 | 4 | 4 | 8 | 2 | 1 |
| Uninitialized data | 29 | 7 | 13 | 5 | 2 | 2 |
| Null dereference | 20 | 9 | 3 | 7 | 1 | 0 |
| Divide by zero | 4 | 2 | 0 | 0 | 1 | 1 |
| Infinite loop | 3 | 1 | 1 | 1 | 0 | 0 |
| Data race / deadlock | 8 | 5 | 1 | 1 | 1 | 0 |
| Memory mismanagement | 10 | 7 | 1 | 1 | 0 | 1 |
| Miscellaneous | 8 | 2 | 0 | 4 | 2 | 0 |
| Total | 141 | 47 | 28 | 35 | 24 | 7 |

**Figure 2: Vulnerabilities (rows) vs. locations (columns).**

information disclosure (CVE-2010-0003), DoS (CVE-2010-2248), or privilege escalation (CVE-2010-3904 and CVE-2010-3081) if the process controls what data to write.

**Missing permission checks.** The kernel performs a privileged operation without checking whether the calling process has the privilege to do so. A vulnerability in this category results in a violation of a kernel security policy. The attacks that can exploit this vulnerability depend on what the security policy is, ranging from arbitrary code execution (CVE-2010-4347), privilege escalation (CVE-2010-2071 and CVE-2010-1146), to overwriting an append-only file (CVE-2010-2066 and CVE-2010-2537).

**Buffer overflow.** The kernel incorrectly checks the upper or lower bound when accessing a buffer (CVE-2011-1010), allocates a smaller buffer than it is supposed to (CVE-2010-2492), uses unsafe string manipulation functions (CVE-2010-1084), or defines local variables which are too large for the kernel stack (CVE-2010-3848). The attacks that can exploit this vulnerability are memory-corruption (for writes) or information-disclosure (for reads) attacks. An adversary can mount privilege-escalation attacks by overwriting nearby function pointers and subverting the kernel's control flow integrity.

**Integer overflow.** The kernel performs an integer operation incorrectly, resulting in an integer overflow, underflow, or sign error. The adversary can trick the kernel into using the incorrect value to allocate or access memory, allowing similar attacks as allowed by "buffer overflow" vulnerabilities. For example, overflow after multiplication can cause the kernel to allocate a smaller-than-needed buffer (CVE-2010-3442); underflow after subtraction can cause memory corruption beyond the end of a buffer (CVE-2010-3873); and sign errors during comparison can bypass bounds checking and cause information disclosure (CVE-2010-3437).

**Uninitialized data.** The kernel copies the contents of a kernel buffer to user space without zeroing unused fields, thus leaking potentially sensitive information to user processes, such as variables on the kernel stack (CVE-2010-3876). This category has 29 vulnerabilities, the highest of all categories. A direct attack using this vulnerability results in unintended information disclosure. However, vulnerabilities in this category may enable other attacks, such as attacks that require knowing the exact address of some kernel data structure, private kernel keys, or other kernel randomness.

**Memory mismanagement.** This category includes vulnerabilities in kernel memory management, such as extraneous memory consumption (CVE-2011-0999), memory leak (CVE-2010-4249), double free (CVE-2010-3080), and use-after-free errors (CVE-2010-4169 and CVE-2010-1188). For the vulnerabilities that we examined, an adversary can mount DoS attacks by exploiting them, although in general arbitrary memory corruption may be possible.

**Miscellaneous.** There are other types of vulnerabilities that usually result in either process crashes, kernel panics, or hangs, such as null pointer dereferences, divide by zeros (CVE-2011-1012 and CVE-2010-4165), infinite loops (CVE-2011-1083 and CVE-2010-1086), deadlocks (CVE-2010-4161), and data races (CVE-2010-4526 and CVE-2010-4248).

One observation from the Figure 1 is that buffer and integer overflows are the top threats to the kernel's integrity: 78% of memory corruption exploits are caused by these two vulnerabilities. This observation is consistent with the report by Christey and Martin [8].

Figure 2 shows the distribution of the vulnerabilities in the Linux kernel source code tree, namely, the code statically linked into the kernel image ("core"), device drivers (`drivers`), network protocols (`net`), file systems (`fs`), and the sound subsystem (`sound`). We observe that a non-trivial portion $(1/3)$ of vulnerabilities are located in the core kernel, while $2/3$ are in loadable kernel modules. Less than 20% of vulnerabilities that we examined are in device drivers.

# 3. STATE-OF-THE-ART PREVENTION

We examined several state-of-the-art kernel security tools to see how many vulnerabilities they can prevent. Figure 3 summarizes the results. The rest of the section discusses each tool and the vulnerabilities or attacks that it can prevent. Our examination mainly focuses on tools that target the OS kernel, and is based on our understanding of these techniques.

## 3.1 Runtime tools

**Software fault isolation.** BGI [6] is a tool to isolate kernel modules with support for controlled sharing between kernel and modules. BGI provides a memory access control list (ACL) for each module. Programmers using BGI set the ACLs, granting or revoking a module's privileges as it invokes various functions in the core kernel (e.g., granting access when allocating memory, and revoking access when freeing memory). In this way, BGI can prevent a vulnerable module from overwriting kernel memory that it shouldn't have access to, such as double-free bugs and some buffer overflows, but allow access to kernel memory that it should have access to.

A major shortcoming of BGI is that it handles only vulnerabilities inside a module, *and* certain attacks that attempt to cross the boundary of the buggy module. As Figure 2 shows, $1/3$ of all vulnerabilities are in the core kernel, and would not be handled by BGI. Moreover, some exploits occur entirely *within* a module, and thus would not be handled by BGI either. For example, if there is a buffer overflow vulnerability in a file system module, the attacker could tamper with the module's internal data, trick the module into executing code that only applies to `setuid` binaries, and gain root privilege. Isolating modules from one another would not solve this problem.

**Code integrity.** SecVisor [22] enforces code integrity for the kernel. A thin hypervisor layer authenticates all code before that code is allowed to execute in kernel mode. SecVisor is effective at preventing code injection attacks. However, although it may defend against common exploits that are unaware of SecVisor's existence, persistent adversaries could still mount attacks by corrupting important kernel data, or by hijacking control flow to existing kernel code. It has been shown that it is possible to use static analysis to combine existing code sequences to perform arbitrary computation [23]. Thus, SecVisor makes exploits harder to write, but does not prevent an adversary from exploiting vulnerabilities, which is why we list zeros in the "SecVisor" column in Figure 3.

**User-level drivers.** SUD [5] runs device drivers at user level and prevents vulnerabilities in the driver from affecting the rest of the kernel. This turns most vulnerabilities in the driver into a denial-of-service attack that crashes the driver itself; a separate recovery mechanism, such as shadow drivers [24], is needed to mitigate the DoS impact. Note, however, that only a small fraction of kernel security vulnerabilities come from device drivers (20% in Figure 2) and that user-level drivers don't address the other vulnerabilities.

**Memory tagging.** Memory tagging systems, such as Raksha [10], can detect when kernel code misuses untrusted inputs (from user processes or from the network), preventing an adversary from mounting code injection attacks or otherwise taking over control flow. As with SecVisor, this prevents certain types of exploits that rely on taking over kernel control flow, but doesn't address many other vulnerabilities in Figure 2.

**Uninitialized memory tracking.** Two systems specifically address the problem of kernel code leaking sensitive data through uninitialized memory. Kmemcheck [20] is a runtime tool in the Linux kernel that detects uninitialized memory vulnerabilities for a given workload. Kmemcheck tracks initialization status of each memory byte, with the help of the kernel memory allocator. Kmemcheck cannot detect reading of uninitialized data from the stack. Secure deallocation [17] (`SD`), on the other hand, periodically zeros out the kernel stack to reduce information leaks from uninitialized stack variables, but does not address dynamically-allocated objects. Neither of these two tools can guarantee that they find and prevent all information disclosure bugs, although they make the bugs more difficult to exploit.

## 3.2 Compile-time tools

In principle, code analysis tools can pinpoint vulnerabilities so that they can be fixed once and for all by developers, and ideally prove the absence of any vulnerabilities in code. Many such tools have been used to find and fix a wide range of security problems in the Linux kernel [1, 2, 4, 13, 21]. One limitation of most static analysis tools is the large number of false positives. Thus, since it is not productive for programmers to filter out these false positives and fix all real vulnerabilities, almost no static analysis tool can prove the absence of vulnerabilities of any type in the Linux kernel, and the tools are largely used for bug-finding. To reduce the number of false positives, many static analysis tools require programmers to supply annotations [1, 2].

One specific class of vulnerabilities that seem to be difficult to detect using static analysis are the semantic vulnerabilities, such as missing permission check bugs. On the other hand, several tools have been effective at finding potential null dereference, buffer overflow, deadlock, and infinite loop bugs [1, 2, 9, 11, 26].

# 4. OPEN PROBLEMS

The examination in the previous section suggests there are several unaddressed challenges facing researchers in dealing with vulnerabilities found in the Linux kernel. First, vulnerabilities are present in almost all parts of the kernel, including device drivers, kernel modules, and core kernel code. Solutions that focus on only one part of the kernel, such as kernel modules, are insufficient by themselves.

Second, many runtime tools focus on preventing a certain class of *exploits*, as opposed to preventing *any* exploit of a certain class of vulnerabilities. The difference is important, since an adversary that knows of a given vulnerability is free to choose any exploit that will work on the target system. Thus, defense mechanisms that still allow a vulnerability to be exploited in some way are of limited use.

Third, many vulnerabilities continue to stem from the fact that Linux is written in an unsafe language. While it might be a good idea to write any new kernel in a type-safe language, we must contend with C, if we want to handle existing Linux code. Similarly, alternative kernel designs, such as HiStar [27] or Minix [16], can significantly reduce the amount of trusted code in the kernel, but these techniques cannot be incrementally applied to Linux. Systems like Overshadow [7] and Proxos [25] can also avoid trusting the entire Linux kernel, for applications that require only a subset of the kernel's functionality, but this approach has not yet been shown to work for all applications. How to incrementally provide type safety and module isolation for large monolithic kernels written in an unsafe language remains an open topic.

Although these challenges are unaddressed, others have identified them too. The rest of this section expands on several challenges that have received less attention.

## 4.1 Semantic vulnerabilities

Figure 3 shows that none of previous research addresses "missing permission checks" vulnerabilities. Unfortunately, these vulnerabilities can easily be exploited to gain privilege. Figure 4 shows a patch to CVE-2010-2071, in which the `btrfs` module forgets to check the owner of a file when setting file permissions. Thus, an adversary can gain write access to any file in the volume. Fur-

| Vulnerability | BGI | SecVisor | SUD | Raksha | kmemcheck | SD |
|---|---|---|---|---|---|---|
| Missing pointer check | 0 | 0 | 3 | 0 | 0 | 0 |
| Missing permission check | 0 | 0 | 1 | 0 | 0 | 0 |
| Buffer overflow | 1 | 0 | 1 | 0 | 0 | 0 |
| Integer overflow (D) | 0 | 0 | 1 | 0 | 0 | 0 |
| Integer overflow (I) | 0 | 0 | 1 | 0 | 0 | 0 |
| Integer overflow (E) | 0 | 0 | 3 | 0 | 0 | 0 |
| Uninitialized data | 0 | 0 | 13 | 0 | 1 | 23 |
| Null dereference | 11 | 0 | 3 | 0 | 0 | 0 |
| Divide by zero | 2 | 0 | 0 | 0 | 0 | 0 |
| Infinite loop | 0 | 0 | 1 | 0 | 0 | 0 |
| Data race / deadlock | 0 | 0 | 1 | 0 | 0 | 0 |
| Memory mismanagement | 1 | 0 | 1 | 0 | 0 | 0 |
| Miscellaneous | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 3: Number of vulnerabilities that existing runtime tools can prevent. For "integer overflow", "D" means that the vulnerability can only lead to DoS attack, "I" means that the vulnerability can only lead to information disclosure, and "E" means that the vulnerability can lead to root privilege escalation, thus any other type of attacks. If a tool can prevent "integer overflow (E)", then for the same vulnerability, it is not credited for "integer overflow (I)" or "integer overflow (D)".**

```
1 --- a/fs/btrfs/acl.c
2 +++ b/fs/btrfs/acl.c
3 @@ -160,3 +160,6 @@ static int btrfs_xattr_acl_set(...
4         int ret;
5         struct posix_acl *acl = NULL;
6
7 +       if (!is_owner_or_cap(dentry->d_inode))
8 +               return -EPERM;
9 +
```

**Figure 4: Patch for CVE-2010-2071 in `btrfs`.**

thermore, the adversary can obtain root access by replacing legal `setuid` executables with malicious ones. Ironically, an identical vulnerability (CVE-2010-1641) was discovered in the `gfs2` module 15 days earlier. Similar vulnerabilities exist in other file systems as well. For example, in CVE-2010-2066, the `ext4` module allows local users to overwrite an append-only file. An attacker can exploit this vulnerability to tamper with audit logs.

In another two semantic vulnerabilities, the kernel permissively exports a sensitive interface to all users, allowing unprivileged users to alter crucial system state. In CVE-2010-1146, for example, the `reiserfs` driver does not prevent the `.reiserfs_priv` directory from being opened, which contains meta-data that should be private to the file system. As a result, unprivileged users could tamper with the exposed meta-data and directly modify ACLs belonging to other users, including root, or set the `CAP_SETUID` extended attribute on a malicious executable to gain privilege.

Another example is CVE-2010-4347, in which the `acpi` module creates a `custom_method` file in `debugfs` with writable permission for all users. The file exports an interface that allows users to define custom methods for the ACPI module to call. Therefore, an unprivileged user could modify the kernel's control flow by writing a specially-crafted value to this file, and gain root privileges.

These vulnerabilities are difficult to check for and defend against, because the security policies are not explicitly stated, and it is usually the programmer's responsibility to enforce them by manually inserting the right checks in each code path. In contrast, the policy for enforcing memory or type safety can often be inferred from the code in a semi-automated fashion.

One possible approach for dealing with these vulnerabilities is based on the observation that there are common interfaces where high-level policies can be checked, or where the policies can be inferred from. For example, in Linux, many file systems implement the VFS interface, and each file system internally is supposed to perform the same set of permission checks (e.g., the permissions on a file can be changed only by the file's owner or by root). A kernel developer could annotate the VFS interface with these high-level policies, and use some type of analysis to check whether every implementing file system performs sufficient checks. Alternatively, an automated analysis could try to infer the set of policy checks needed, by comparing many file system implementations to each other, similar to the idea of detecting bugs as deviant behavior [14].

Another possible approach to dealing with these bugs may be to reason about privilege separation in terms of user or application principals, instead of kernel module principals (as most current kernel isolation systems have done [5, 6, 15, 16]). While user boundaries may not correspond to clean boundaries in kernel code, if one can enforce isolation in terms of user privileges, one may not need to reason about vulnerable modules, as in Loki [28].

## 4.2 Denial-of-service vulnerabilities

Few previous pieces of work address vulnerabilities that lead to only denial-of-service (DoS) attacks, possibly because DoS attacks do not harm the data integrity of the kernel. However, as pointed out by an earlier study [3], any kernel weakness can turn out to be a security threat. For example, the `econet` privilege escalation exploit involved three separate vulnerabilities that were not originally considered to be security-critical [12], including one DoS vulnerability.

Dealing with denial-of-service vulnerabilities in kernel code is difficult, because it requires not only detecting and preventing an adversary from crashing the kernel, but also requires continuing the kernel's execution to properly perform operations on behalf of other users. This can be particularly difficult if the denial-of-service vulnerability is triggered while a kernel thread is holding locks, or otherwise cannot be terminated cleanly without violating kernel invariants. One approach to ensuring the kernel continues to operate despite denial-of-service vulnerabilities is to use shadow drivers [24] or recovery domains [18], but no general-purpose techniques for ensuring availability despite these vulnerabilities are available.

# 5. SUMMARY

This paper's study of Linux kernel vulnerabilities suggests that we have a long way to go in making existing OS kernels secure. First, from January 2010 to March 2011, 141 vulnerabilities in the Linux kernel were discovered, many of which have serious exploits. Second, state-of-the-art defense techniques address only a small subset of them. Third, some of the unaddressed vulnerabilities, such as semantic bugs, pose challenging research problems.

## Acknowledgments

We thank the anonymous reviewers for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089. The opinions in this paper don't necessarily represent DARPA or official US policy.

## References

[1] Smatch. `http://smatch.sourceforge.net/`.

[2] Sparse. `http://sparse.wiki.kernel.org/`.

[3] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security impact ratings considered harmful. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[4] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2002.

[5] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, June 2010.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008.

[8] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. `http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf`, 2007.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 19th Usenix Security Symposium*, San Jose, CA, July 2008.

[11] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[12] N. Elhage. CVE-2010-4258: Turning denial-of-service into privilege escalation. `http://blog.nelhage.com/2010/12/cve-2010-4258-from-dos-to-privesc/`, 2010.

[13] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[15] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[17] C. Jim, P. Ben, G. Tal, and R. Mendel. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th Usenix Security Symposium*, Baltimore, MD, August 2005.

[18] A. Lenharth, V. S. Adve, and S. T. King. Recovery domains: an organizing principle for recoverable operating systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.

[19] S. A. Mokhov, M.-A. Laverdiere, and D. Benredjem. Taxonomy of Linux kernel vulnerability solutions. *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*, 2008.

[20] V. Nossum. Getting started with kmemcheck. `http://www.mjmwired.net/kernel/Documentation/kmemcheck.txt`.

[21] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, March 2011.

[22] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[23] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, October–November 2007.

[24] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[25] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[26] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32rd ACM Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005.

[27] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[28] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

# Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters*

Liwei Yuan, Weichao Xing, Haibo Chen, Binyu Zang
Parallel Processing Institute
Fudan University
{yuanliwei, wcxing, hbchen, byzang}@fudan.edu.cn

## ABSTRACT

This paper considers and validates the applicability of leveraging pervasively-available performance counters for detecting and reasoning about security breaches. Our key observation is that many security breaches, which typically cause abnormal control flow, usually incur precisely identifiable deviation in performance samples captured by processors. Based on this observation, we implement a prototype system called Eunomia, which is the first non-intrusive system that can detect emerging attacks based on return-oriented programming without any changes to applications (either source or binary code) or special-purpose hardware. Our security evaluation shows that Eunomia can detect some realistic attacks including *code-injection attacks*, *return-to-libc attacks* and *return-oriented programming attacks* on unmodified binaries with relatively low overhead.

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

## General Terms

Security

## Keywords

Performance Counters, Return-oriented Attacks, Return-to-libc Attacks, PMU Deviation

## 1. INTRODUCTION

Security breaches are a major threat to the dependability of computer systems and can cause not only economic effect but also social impact. However, many previous mitigation approaches are either heavyweight in requiring non-trivial modifications to processor architectures [7], or intrusive in requiring source-code or binary modifications to applications [15]. Further, some emerging attacking approaches such as return-oriented programming [11] are usually difficult to be detected using existing techniques.

In this paper, we propose a non-intrusive approach to detecting a series of security breaches, which requires no changes to existing hardware, source code or binaries. The approach we propose, namely Eunomia, leverages the pervasively available performance monitoring units (PMUs) in commercial processors, to monitor performance deviation of a running application to detect possible attacks.

The key observation is that security breaches usually cause abnormal control flow that can be precisely captured by PMUs in processors. For example, attacks involving code-injection usually need to transfer control flow to injected code, while return-to-libc attacks and return-oriented programming need to break the control flow integrity of a program.

To validate our observation, we design and implement a prototype called Eunomia, to detect a series of attacks, including both code-injection and return-based[1] attacks. To detect such attacks, Eunomia continuously monitors performance samples using specific events during the program execution and correlates the events with the program execution context (such as instruction addresses and callers of library functions). Any deviation in performance samples can be used as signs of possible attacks. Upon the detection of an attack, the performance samples information such as *branch traces* can be used to locate the exploited security vulnerability and determine how the vulnerability is exploited.

To confirm the effectiveness of Eunomia, we have conducted some preliminary security tests using real-world vulnerabilities. Our evaluation results indicate that Eunomia can precisely detect the attacks at the first time they happen. Performance evaluation results show that Eunomia incurs small performance overhead for real-world applications.

## 2. SECURITY BREACHES AS PMU DEVIATION

### 2.1 Candidate PMU Features

There are a number of standard PMU features such as recording the occurrences of branch miss predictions, instruction TLB (iTLB) misses and L2 cache misses. Further, recent processors also provide several advanced features. The followings list two features from recent Intel processors:

*This work was funded by China National Natural Science Foundation under grant numbered 90818015 and 61003002, and a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100.


*APSys'11,* July 11–12, 2011, Shanghai, China.


[1]We consider return-to-libc attack and return-oriented programming attack as return-based attacks

**Precise Event Based Sampling (PEBS):** In PEBS, performance samples are directly recorded into a predefined memory region when the occurrences of the event exceed a threshold, instead of generating an interrupt immediately. An interrupt is generated when predefined memory region is full. This mechanism enhances performance monitoring performance by batching each sample into a predefined buffer and processing them together. With the *atomic-freeze feature*, the reported IP in predefined memory is precisely the instruction immediately following the instruction causing the event.

**Branch Trace Store:** BTS provides a mechanism to capture control transfer events including all types of jump, call, return, interrupt and exception. The recorded information includes the source address, target address and properties of the control transfer. Thus, it provides the ability to trace the control flow of program execution. In Intel Core and Core i7, the branch trace information is generated and delivered to the system bus, and then directly recorded into a predefined memory region, similar to PEBS.

## 2.2 PMU Deviation in Common Attacks

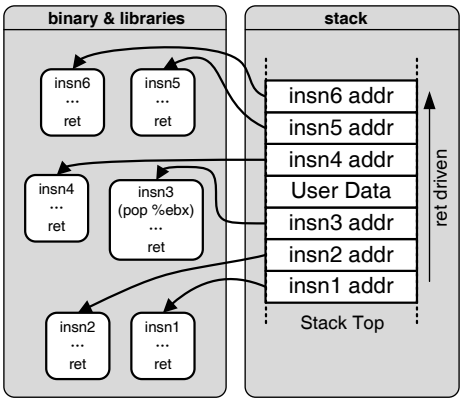


**Figure 1: An example illustrating return-oriented programming.**

Currently, there are two typical types of attacks according to the means to gain control of a program: code-injection attack, which directly injects external shellcode to a program; and return-based attack, which reuses existing binary code as shellcode. Code-injection attacks usually inject external shellcode to a program by exploiting some vulnerabilities, such as buffer overflow, format string and dangling pointer, and then overwriting a return address or a function pointer to transfer control to the injected code.

For return-based attacks, we further divide them into two types: return-to-libc attacks and return-oriented programming attacks. Return-to-libc attacks directly exploit security-sensitive library routines such as "execve" with attacker-supplied arguments (e.g., a root shell) to gain control. By contrast, return-oriented programming reuses binary code fragments in existing binary to form a shellcode. The essence of return-oriented programming is illustrated in Figure 1, which leverages control of the call stack to indirectly execute cherry-picked machine instructions or groups of machine instructions immediately prior to the "ret" in subroutines within existing binary. The composed code may act as a typical shellcode that binds a tcp to open a remote root shell upon connection.

According to our analysis, these attacks above, no matter how they exploit vulnerabilities (e.g., format string or buffer overflow), usually behave different in performance samples. Hence, unlike some prior attack detection schemes that focus on a particular security vulnerability, our method detects attacks not according to which vulnerability they exploit, but how they behave abnormally in performance samples. We find that both the two types of attacks result in deviation in control flow, thus incur some precisely identifiable performance samples. Table 1 summarizes the performance deviation in such attacks and how the attacks can be monitored with PMU events. As the table shown, there are some PMU events to detect each type of these attacks. For code-injection attacks, the injected code is executed on data sections (e.g., stack or heap), thus results in deviation for program counters in *branch_miss_predict*, *itlb_misses* and *branch traces*. For those reusing library routines or binary code, they will result in abnormal control flow in attacking runs that violate the control flow integrity. This will be reflected by abnormal performance samples using the BTS mechanism in x86 architecture.

## 2.3 Detecting Attacks Using Performance Counters

**Detecting Code-injection Attacks:** For attacks that inject shellcode and cause control transfer to the injected code in data sections (e.g., stack or heap), there will be abnormal performance samples such as *itlb_misses* in data sections. This provides a strong evidence of possible attacks.

Using this method to defend against code-injection attacks has similar protection ability with non-execution bit protection, but it is more flexible. Eunomia can do application-specific pattern analysis, to identify the patterns of how an application executes code in data sections and filter them out.

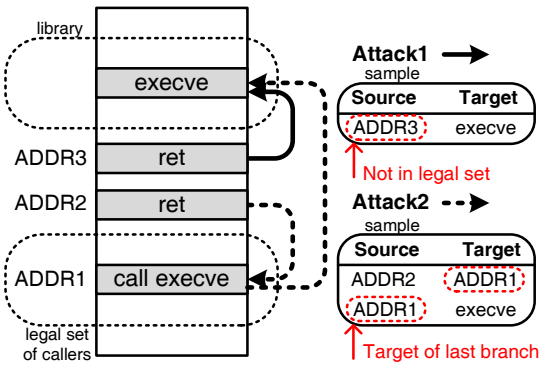


**Figure 2: The algorithm to detect return-to-libc attacks.**

**Detecting Return-to-libc Attacks:** For attacks that directly reuse sensitive library routines such as "execve" with attacker-supplied arguments, we need to check the callers of this library routine. Before running a program, Eunomia preprocess application binaries, analyzes the callers of library routines and records callers for each library routine as its legal callers, as shown in Figure 2. As library routines are called from GOT entry[2] whose entry addresses are statically determined, Eunomia can easily determine the legal callers of each library routine. We only need to record legal callers in applications.

During the check, Eunomia uses the branch trace store (BTS) mechanism to record each control transfer and checks the source and target of each transfer in the sample. If the branch target is a library routine, Eunomia checks if the source address is in the legal

[2]GOT entry is short for global offset table, which works together with program linkage table to identify loaded library routine address in dynamic linker.

| Attack Type | Description | PMU Events |
|---|---|---|
| **Code-injection** | Inject code and take control transfer to injected code | Branch Tracing Event(BTS)<br>Branch Miss Predict Event<br>Instruction TLB Misses Event |
| **Return-to-libc** | Use library calls instead of inject code (e.g., invoke "execve" with "bin/bash") | Branch Tracing Event(BTS) |
| **Return-oriented programming** | Use instructions before "ret" in existing library and binary code to form shellcode | Branch Tracing Event(BTS) |

**Table 1: Deviation in performance characteristics of common attacks.**

set of callers and reports an alarm if not. However, attacks can use existing "call" in binary to indirectly invoke a library routine. For such attacks, Eunomia checks one more step further. If the target address in previous sample and the source address in this sample are the same, Eunomia reports an alarm, as illustrated in Figure 2. This method may cause false positives if there are consecutive calls in normal execution, since it behaves like indirect caller of library routine in samples. To filter out this kind of false positive, Eunomia records these consecutive calls when doing binary preprocessing and avoids reporting an alarm in this situation.

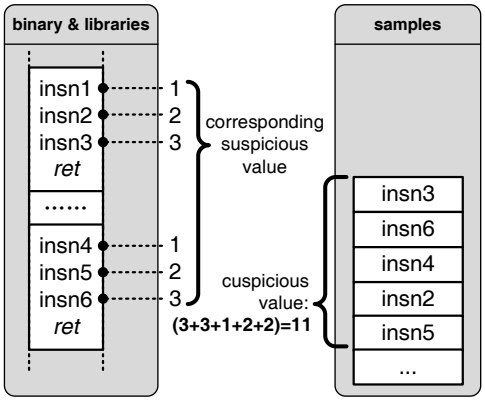


**Figure 3: The algorithm to detect return-oriented programming attacks, which uses suspicious value of insns before "ret" and checks the accumulated value against the predefined threshold.**

**Detecting Return-oriented Programming Attacks:** To detect return-oriented programming attacks, Eunomia needs to check the control flow integrity of a program execution. Ideally, Eunomia should record all legal addresses of a "ret" instruction can return to. However, due to the "unintended instruction sequences" in x86, all opcode has the byte "0xc3" could be viewed as the instruction "ret", even if the byte "0xc3" is in a jump instruction (e.g. "jmp 0x3aae9" has opcode "e9 c3 f8 ff ff"). It is usually difficult and time-consuming to build and check with the legal set of control flow graph.

Because most of such attacks need to reuse several instructions before instruction "ret" to form shellcode [14], the target of branches in return-oriented shellcode typically follows a "ret" instruction within usually one to three instructions [3].

Hence, Eunomia exploits the branch tracing ability provided by BTS to detect such attacks. Before running a program, we preprocess the binary, analyze all instructions that appear before "ret"[4]. We assign a suspicious value to each instruction according to their distance to "ret" and store the IP, opcode and the suspicious value into a hashtable, as shown in Figure 3. For dynamically linked libraries, Eunomia also processes them when the dynamic linker loads libraries and updates the hashtable accordingly. During program execution, Eunomia checks the address of each branch target in performance samples to see whether it is in the hashtable. If a consecutive number of branch targets fall into the hashtable and the accumulated suspicious value exceeds a predefined threshold, a return-oriented programming attack has likely occurred. To our knowledge, Eunomia is the first system that can detect return-oriented programming attack without modification to programs [12], binary instrumentation [8] or special-purpose hardware.

Currently, Eunomia does not consider return-oriented programming that uses an update-load-branch sequence [4] such as *pop %eax; jmp ∗%eax*. As the BTS mechanism supports collecting all branch traces, Eunomia should be able to extend Eunomia to detect such a scheme. Eunomia could be enhanced to collect all instructions with *update-load-branch* effect and use the collected branch traces to identify abnormal control transfers, which will be our future work.

## 3. IMPLEMENTATION OF Eunomia

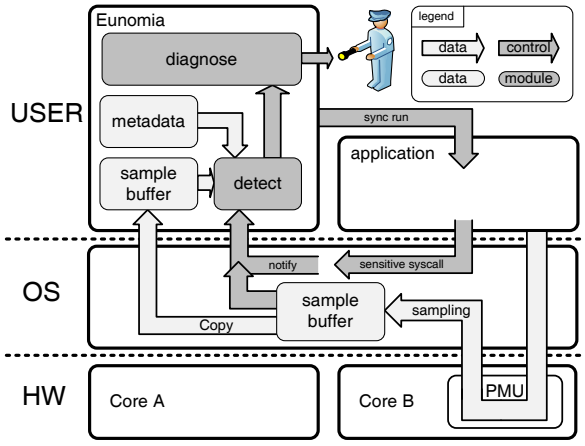


**Figure 4: Architecture of Eunomia.**

We have implemented a preliminary version of Eunomia based on *perf_events* on Linux kernel version 2.6.34. Currently, Eunomia supports Intel Core Duo and Core i7 processors and focuses on user-level attacks only.

The overall architecture of Eunomia is shown in Figure 4. Eunomia is composed of two parts: the kernel extension that provides interfaces for user tools to set up events and read performance samples; and a user-level tool that uses the performance

[3]Those reusing more than 3 instructions before "ret" are hard or impossible to construct in real-world [14].

[4]This "ret" refers to all "0xc3" including "ret" in "unintended instruction sequence".

samples to detect possible security attacks. The user-level tool is in the form of a monitoring process, which checks the performance samples generated by the application processes. To synchronize the monitoring process with the application processes, the monitoring process is executed as the parent process of the application processes. Eunomia uses ptrace-based techniques to synchronize monitoring process and application processes. To start an application, the monitoring process forks a child process. Before the child uses *exec* to run the application, the child process calls *ptrace* with the *PTRACE_TRACEME* flag to cause the child process to be suspended when *exec* is executed. The monitoring process sets up performance events and sample buffers for the child process and then lets the child process run. Afterwards, these two processes run simultaneously. To reduce interference between the monitoring process and the application processes, Eunomia supports the binding of each process into different cores on multicore hardware.

During normal execution, the event samples of the application processes are written to a performance sample buffer monitored by the monitoring process. As the per-sample check is very costly, Eunomia chooses to check samples in a batched mode, which notably reduces performance overhead. Fortunately, in Intel processor families, they widely use predefined buffer to record events (e.g., PEBS and BTS), this mechanism not only amortizes the cost of exception handling, but also enhances the accuracy of sampled information and fits naturally into our batch-mode checking. For events without predefined buffers (e.g., *itlb_misses* and *branch_miss_predict*), Eunomia manually records the performance samples into a buffer and signals the monitoring process when the buffer is full. Eunomia controls the frequency of security checks by setting the length of the sample buffer.

The monitoring process is executed in an event-driven way and will take action in two cases: a signal is received (e.g., the event buffer is full); or the application processes are trying to invoke sensitive system calls (e.g., execve, which is usually called by shellcode). In both cases, the monitoring process will suspend the application processes, do the security check and then resume the application processes after the check. This prevents the application processes from running out-of-sync, which may cause the sample buffer being overwritten with new data, or harmful effects to the system being made by attacks. According to our experience, the checking time in the monitoring process is very short with low CPU utilization. Thus, the overhead due to the suspend time of application processes is very small.

Eunomia also supports the post-attack analysis by using the performance samples of BTS, which provides useful information for detailed post-attack diagnosis. Thus, Eunomia is capable of finding the trace from normal function to the injected code. To use performance samples for post-attack diagnosis, Eunomia can dump a portion of performance samples containing abnormal control flow.

# 4. PRELIMINARY RESULTS

All evaluations were performed on an Intel Core i7 processor with 4 cores. The operating system is a Redhat Enterprise Linux with kernel version 2.6.34.

## 4.1 Security Analysis

We use Samba Server version 3.0.21 with a heap overflow vulnerability (CVE-2007-2446) and Squid-2.5.STABLE1 with a stack overflow vulnerability (CVE-2004-0541) to illustrate the detectability of Eunomia. We attack them by three means, namely code-injection attack, return-to-libc attack and return-oriented programming attack. However, due to the vulnerability type, we can not form a return stack in Samba Server, so we failed to exploit Samba Server with return-oriented programming attack. To detect these attacks, we set the threshold of performance events to *one* [5] to minimize false negatives. The threshold for detecting return-oriented programming attacks is set as *15*, which is enough to detect most attacks. As expected, all attacks are detected by Eunomia in the evaluation and Eunomia detects these attacks at the first time an abnormal performance sample is generated.

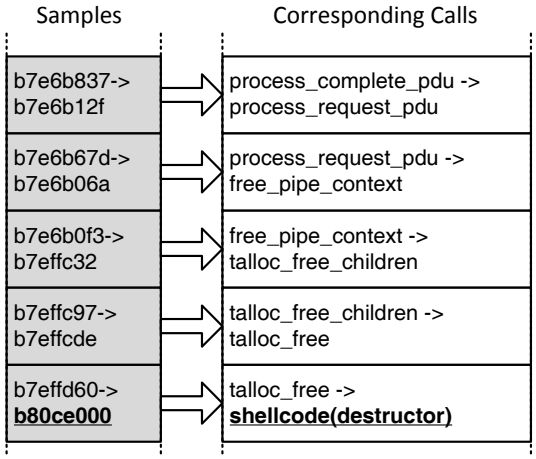


**Figure 5: The results of post-attack diagnosis of code-injection attack for Samba Server.**

**Post-Attack Diagnosis:** We also use Samba Server to illustrate the post-attack diagnosis ability of Eunomia using the BTS mechanism. As shown in Figure 5, when a code-injection attack is detected, Eunomia dumps the performance samples with abnormal control flow. From the back traces of the attack, we can easily find that the shellcode is reached by calling "destructor" function pointer in "talloc_free". Virtually, Eunomia can back trace as far as the dumped samples can reach. Here, we only provide five function records in the back trace, which is usually enough for understanding the attack.

## 4.2 Performance Results

We evaluate several applications in three categories: widely-used server applications, including Apache Web Server, Postgres Database Server and Exim Mail Server [1]; emerging applications including Memcached [10] and Metis [13]; and daily used tools in Linux system, including OpenSSH, a server application for secure remote shell and Tar, which is a compressing tool. For Apache Web Server and OpenSSH, we evaluate them by both latency and throughput. The latency for OpenSSH is the connection time, which is measured by calculating the average connection time of 10,000 connections. The throughput for OpenSSH is measured by transferring 1 Gbyte file using *scp*. For Apache Web Server, we use apache benchmark (*ab*) by issuing 500,000 requests with concurrency level of 100.

**Performance for Detecting Code-injection Attacks:** There are three events that can be used to detecting code-injection attacks, which are *itlb_misses*, *branch_miss_predict* and BTS mechanism. According to our evaluation, the event counts of *branch_miss_predict* could be more than 10 times larger than the counts of *itlb_misses* and control transfer counts captured by BTS mechanism could be 80 times larger than the counts of *itlb_misses* in the same run. Using benchmarks mentioned above to evaluate these three types of events, we found relative performance

[5]The threshold for BTS is one in nature.

overhead is 4.72%, 1% and 0.1% on average for using BTS, *branch_miss_predict* and *itlb_misses* accordingly. Thus, for detecting code-injection attacks, *itlb_misses* event is most appropriate.

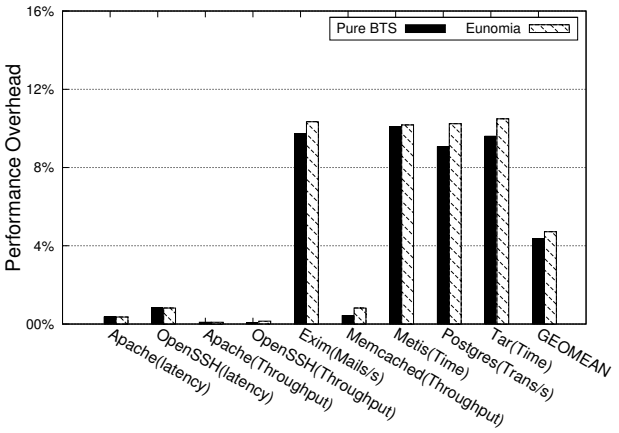


**Figure 6: Performance overhead of Eunomia to detect code-injection and return-based attacks with BTS.**

**Performance for Attack Detection with BTS:** Figure 6 shows the relative performance overhead for these applications, from the figure, we can see that Eunomia incurs acceptable performance overhead, with only 4.72% on average, ranging from 0.09% to 10.49%, which means Eunomia can be applied to real-world applications on off-the-shell systems in daily use.

We also measured the inherent overhead of the BTS mechanism alone and found that the performance overhead is almost the same as using BTS together with Eunomia, as shown in Figure 6. This indicates that the detection logic of Eunomia causes very little performance overhead itself. The major reason for overhead in BTS is due to the large number of performance samples, including *all types of jump*, *exceptions*, *calls* and *returns*. Among them, we actually only need samples for call and returns. This demands a hardware supported samples selection mechanism.

# 5. RELATED WORK

Performance counters have been used extensively for performance profiling and online optimization. Being aware of the importance of performance counters, previous researchers have proposed a variety of architectural techniques in order to provide low-overhead, non-intrusive and accurate performance monitoring [9, 2]. Avritzer et al. [3] performed a set of tests to measure CPU, memory and I/O usages between normal and attacking runs and concluded that the *accumulated resource usages* tend to be different. However, they failed to show how to leverage the difference for precise and in-place detection of attacks.

As Eunomia, previous researchers have also leveraged existing hardware support for security. For example, SHIFT [5] exploits existing hardware support for control speculation to implement an efficient and flexible taint tracking system. BOSH [6] uses the flow-sensitive tags in taint tracking to implement an efficient binary obfuscation system. Compared to Eunomia, these systems require instrumenting the software using compilers, thus cannot work on unmodified and deployed binaries.

# 6. CONCLUSION AND FUTURE WORK

This paper observed that typical security exploits usually result in precisely identifiable deviation in performance samples. Based on the observation, we designed and implemented Eunomia, which leveraged performance counters for the purpose of detecting a wide variety of security attacks. Our evaluation showed that Eunomia could effectively detect attacks on real-world security vulnerabilities with low overhead. Our future work includes more analysis on false positives and false negatives in Eunomia, enhancing existing PMUs to detect more security vulnerabilities as well as software bugs, and cooperating with compiler transformations or instrumentation to increase the precision of Eunomia.

# 7. REFERENCES


[1] Exim. http://www.exim.org/.

[2] AMD. Instruction-based sampling: A new performance analysis technique. developer.amd.com/assets/amd_ibs_paper_en.pdf.

[3] A. Avritzer, R. Tanikella, K. James, R. G. Cole, and E. Weyuker. Monitoring for security intrusion using performance signatures. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 93–104, 2010.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. CCS*, 2010.

[5] H. Chen, X. Wu, L. Yuan, B. Zang, P. Yew, and F. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *Proc. ISCA*, pages 401–412, 2008.

[6] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P. Yew. Control flow obfuscation with information flow tracking. In *Proc. MICRO*, pages 391–400, 2009.

[7] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. MICRO*, 2004.

[8] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proc. ACM workshop on Scalable trusted computing*, pages 49–54, 2009.

[9] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. MICRO*, pages 292–302, 1997.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[11] R. Hund, T. Holz, and F. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proc. Usenix Security*, pages 383–98, 2009.

[12] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with Return-Less kernels. In *Proc. Eurosys*, pages 195–208, 2010.

[13] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for Multicore Architectures. *MIT-CSAIL-TR-2010-020*, 2010.

[14] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. http://www-cse.ucsd.edu/ hovav/dist/rop.pdf, 2010.

[15] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. Usenix Security*, 2006.

# A Virtual Memory Foundation for Scalable Deterministic Parallelism

Yu Zhang
University of Science and Technology of China
yuzhang@ustc.edu.cn

Bryan Ford
Yale University
bryan.ford@yale.edu

## ABSTRACT

Recent deterministic execution environments promise efficient program replay and bug reproduction, but their scalability is currently limited by strictly hierarchical synchronization models or serialized thread scheduling mechanisms. To address these issues, we introduce a *single-producer multiple-consumer* (SPMC) virtual memory foundation for deterministic parallelism, which supports non-hierarchical synchronization without serialized thread scheduling. An extension to the Determinator microkernel, supporting SPMC memory regions, offers threads and processes scalable "peer-to-peer" communication while preserving the kernel's existing guarantee of system-enforced determinism. DetMP, a deterministic user-level message passing API modeled on MPI, illustrates one way to build convenient application-level parallel programming abstractions atop the SPMC foundation. Preliminary results suggest that DetMP atop SPMC may be realistic and useful, achieving near-ideal speedup for parallel matrix multiplication, and good scaling for IS, in all cases ensuring strict determinism.

## Categories and Subject Descriptors

D.4.1 [**Process Management**]: Multiprocessing/multiprogramming; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages

## General Terms

Design, Languages, Performance

## Keywords

Deterministic Parallelism, Synchronization, Message Passing

## 1. INTRODUCTION

A parallel program is *deterministic* if, for a given input, every execution of the program yields identical behavior and output. Determinism offers benefits for replay debugging [20], fault tolerance [10], and security [2, 14]. Current methods of executing parallel programs deterministically show promise [7, 11, 13, 22], but often incur high costs, allow buggy or malicious applications to defeat repeatability [5], and often do not actually eliminate races from the programming model [3]. All of these approaches face scalability challenges: Grace [8] and Determinator [5] support only hierarchical synchronization such as *fork/join* and *barrier*, for example, while deterministic schedulers offering a full pthreads-compatible API [7, 11, 13, 22] introduce inherently-serial thread coordination mechanisms that make scaling difficult [23].

This paper extends the Determinator microkernel [5] with *single-producer multiple-consumer* (SPMC) virtual memory, a novel operating system foundation for non-hierarchical deterministic communication and synchronization. Any process can create an SPMC memory region, then transfer to other processes a single *producer mapping* of the region, and/or any number of *consumer mappings*, in order to set up virtual "pipes" among processes. A process attempting to touch a page in a consumer mapping blocks until the producer *commits* the corresponding page. Unlike classic Unix pipes, SPMC allows the producer of a region to commit its contents in any order at page granularity, and multiple consumers can read the same pages, efficiently supporting multicast communication patterns common in parallel programming practice.

By conforming to the constraints of the Kahn process model [19], SPMC preserves Determinator's ability to enforce deterministic execution of both multi-threaded and multi-process parallel computations. Because SPMC region transfer still occurs only via direct parent/child interactions between processes and threads, SPMC also preserves the conceptual simplicity, control, and composition power of Determinator's strictly hierarchical "nested process model" [17]. Once SPMC mappings are set up, however, processes can produce and consume pages in these regions incrementally, allowing for arbitrary peer-to-peer "dialog" short-circuiting the process hierarchy, thereby removing Determinator's prior scalability-limiting restriction of requiring all inter-process communication to involve a common ancestor. Finally, by building on the virtual memory capabilities of standard processors, SPMC offers performance characteristics well-matched to modern shared memory multicore machines.

We envision using the SPMC foundation eventually to support both shared memory (SM) and message passing (MP) parallel programming models, as well as high-performance deterministic I/O. As an initial case study, however, this paper introduces and focuses on DetMP, a deterministic message passing API modeled on the well-known MPI framework [21]. Leveraging Determinator's SPMC foundation and the shared memory multicore hardware on which it runs, DetMP offers applications some of the convenience of a shared memory programming model by allowing DetMP programs to share read-only data efficiently without explicit communication. Cooperating application processes use an MPI-like API, restricted to messaging operations with deterministic semantics, to communicate mutable intermediate results.

As a preliminary evaluation of SPMC and DetMP, we examine two well-known benchmarks, matrix multiplication (MM) and integer sorting (IS). Comparing MM atop DetMP against MM on De-


*APSys'11,* July 11-12, 2011, Shanghai, China.

terminator's original hierarchical shared memory model (DetSM), DetMP offers speedup close to ideal, scaling better than DetSM. Comparing IS from NPB-MPI [12] implemented atop DetMP against IS for MPI on Linux, we find that IS-DetMP offers 110% of the speedup of IS-LinuxMPI on 2 CPUs, and 97% and 78% of that of IS-LinuxMPI on 4 and 8 CPUs, respectively, offering good scalability while (unlike Linux) guaranteeing determinism.

This paper makes three main contributions. First, we introduce SPMC, a virtual memory foundation for non-hierarchical deterministic parallelism. Second, atop SPMC we develop DetMP, a deterministic asynchronous message passing API fully compatible with existing languages such as C, and similar to familiar message-passing frameworks such as MPI. Third, our preliminary performance results offer evidence that DetMP atop SPMC may be a realistic and useful approach to deterministic parallel programming.

The remainder of the paper is structured as follows. Section 2 summarizes relevant background work, then Section 3 presents the SPMC model and DetMP. Section 4 examines the prototype and reports preliminary results, and Section 5 concludes.

## 2. BACKGROUND AND RELATED WORK

As the most common forms of parallel programming models, SM and MP models provide different synchronization concepts and constructs, bringing different issues on determinism.

### *Deterministic Shared Memory Programming.*

In the shared memory (SM) model, parallel tasks (usually threads) share an address space to which they read and write concurrently. Although *fork/join* and *barrier* are naturally deterministic synchronization patterns among threads, synchronization mechanisms such as locks and semaphores popularly used in controlling concurrent access are semantically nondeterministic and invite heisenbugs [1].

Deterministic schedulers [7,8,11,13,22] can make bugs reproduciable, but allow misbehaved software to defeat repeatability, and face scalability challenges [23]. DMP [13] and CoreDet [7] interleave threads' synchronization and memory access operations on an artificial schedule. Because this deterministic schedule is semantically arbitrary—not implied by anything in the program's logic—this approach makes race conditions reproducible but does not eliminate them: slight input changes may still reveal schedule-dependent bugs. TERN [11] attempts to address this problem of instability across inputs by reusing past schedules for similar future inputs. Kendo [22] provides deterministic guarantees for data-race-free programs by ordering lock acquisitions and releases. Grace [8] provides deterministic execution for C/C++ fork/join parallel programs using paged-based software transactional memory techniques.

New programming languages and type systems, such as Deterministic Parallel Java [9], offer determinism but require programmers to rewrite legacy code and perhaps adopt unfamiliar concepts.

Rather than designing new languages or deterministic schedulers, the WC memory model [4] allows concurrent threads logically sharing an address space but never seeing each others' writes, except when they synchronize explicitly and deterministically. The experimental Determinator OS [5] is one implementation of WC, but previously supports only strictly hierarchical synchronization.

### *Deterministic Message Passing.*

In the message passing (MP) model, parallel tasks (usually processes) exchange data through by messages to one another, maintaining private memories. Communications among processes are non-hierarchical and may be *asynchronous* (in which the sender does not wait for the receiver to be ready) or *synchronous* (requiring the sender and receiver to wait for each other to transfer a message).

Although MP separates processes' data spaces, it does not automatically offer determinism: results can depend on order of message reception. A message passing API following the constraints of Kahn process networks [19] can offer deterministic execution, but popular MP frameworks such as MPI [21] do not satisfy these constraints. Furthermore, the requirement to marshal and unmarshal data into messages can be both less convenient to programmers and less efficient, even if optimized for shared memory hardware [24].

SHIM [15,16] offers deterministic MP in the Kahn network model, but requires adoption of a new programming language.

## 3. SPMC MODEL AND USAGE

This section describes the proposed SPMC model for deterministic parallelism, first covering the SPMC region primitive provided by the Determinator kernel, then the user-level DetMP message passing API that builds a familiar environment atop this primitive.

### 3.1 SPMC Region Kernel Primitive

Determinator initially constrained inter-process communication to direct parent/child relationships [5]. To alleviate this limitation, we extend Determinator's virtual memory system with *SPMC regions*, which allow a process $P$ to establish communication directly among different children or descendents of $P$, eliminating $P$ as a scalability bottleneck in subsequent computation. The kernel constrains SPMC regions to guarantee determinism despite in the presence of this "peer-to-peer" communication. SPMC regions thus form communication primitives analogous to Kahn process networks [19], expressed in a virtual memory API.

Determinator originally allows only read-only sharing of physical memory, through the kernel's copy-on-write (COW) mechanisms. SPMC generalizes the kernel with a restricted form of read/write sharing, by distinguishing two types of memory mappings. A given physical page can have only one *producer* mapping at a time, in one process's address space. The page may have any number of *consumer* mappings in multiple processes, however. The kernel enforces a protocol in which consumers have no access to a page while the producer is writing to it, but once the producer explicitly *fixes* an SPMC page using a system call described below, the producer loses write access while all consumers gain read access. Figure 1 illustrates a simple scenario in which two processes use two SPMC regions for bidirectional communication.

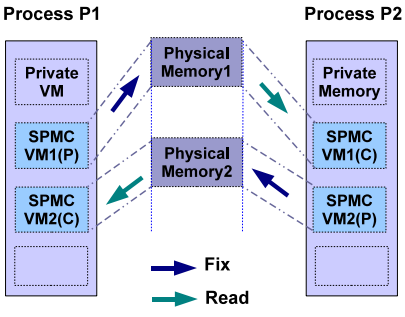


**Figure 1: Processes can share several SPMC virtual memory regions, each of which has only one producer mapping, and any number of consumer mappings. Consumers can access an SPMC page only after the producer explicitly fixes it.**

**Table 1: Extended options to the system calls in Determinator.**

| Put | Get | Option | Description |
|---|---|---|---|
| ✓ | ✓ | Own | Put/Get the ownership of all or part of an SPMC region to/from child. |
| | ✓ | Fix | Fix all or part of an SPMC region. |

To support SPMC regions, we add two optional arguments to Determinator's existing Put/Get/Ret system calls, as shown in Table 1. A user process $P$ calls Get with the Zero and Own options to get a Zero-filled SPMC region, and becomes the owner (producer) of that region. $P$ can transfer ownership of all or part of this SPMC region to any child, by invoking Put with the Own and Copy options; the parent then becomes a consumer of the transferred region. $P$ can also hand out consumer mappings of the region to any number of child processes, by calling Put with the Copy option.

As in Determinator's current semantics, if at the time of a Get/Put call the specified child is still executing, the kernel blocks the parent until the child stops due to a Ret call, processor trap, or deterministic quantum expiration. This cooperative synchronization is necessary for determinism, but occurs only at SPMC region "setup" time. After region setup, the processes holding SPMC mappings can interact and synchronize in a more fine-grained, peer-to-peer fashion, unrestricted by the process hierarchy.

The kernel maps SPMC regions of the producer and all consumers to the same physical memory pages, as shown in Figure 1. To enforce determinism, the kernel never permits consumers to read a memory page while the producer is writing to it. Instead, the kernel gives *only* the producer access to a given page, until the producer explicitly *fixes* the page with a system call. The producer then loses the ability to write to the page, but all consumers gain the ability to read the page, and the page remains read-only for the rest of its lifetime. If a consumer attempts to read an SPMC memory page before the producer fixes it, the kernel blocks the consumer, to be awoken later when the producer fixes that page. Each process can unmap its SPMC regions when it does not need the mapped pages anymore, and the kernel frees each underlying physical page when all the mappings of that page are removed or replaced.

Once an SPMC region has been fixed, a process $P$ can call Get with the Own option to "regain" the ownership of the region. The kernel then remaps the SPMC region of $P$ to other newly-allocated physical pages, then copies the content from the original physical pages to the new ones, so as not to affect other consumers of the original SPMC pages. $P$ can then create new consumer mappings of those pages by invoking Put with the Copy option, or transfer ownership to a child by specifying the Own and Copy options.

With SPMC regions, the kernel thus allows inter-process synchronization to "short-circuit" the process hierarchy while preserving a deterministic execution model. The copy-on-write mechanism optimizes large copies to avoid physically copying read-only pages. We intend this communication model to serve as a deterministic foundation to support multiple deterministic parallel programming models, as well as scalable deterministic I/O capabilities in the future. For the present, however, we focus on using SPMC for message-passing parallel applications, as described next.

## 3.2 DetMP User-Space Parallel API

DetMP is a user-space library built atop the kernel's SPMC region primitive, offering a high-level *channel* abstraction for message passing, and an MPI-like set of collective communication operations.

A channel is a pipe-like abstraction implemented and managed in user space by the DetMP library. Each channel has a globally-unique ID, `cino`, and has a unique producer and one or more consumers. The producer can asynchronously send several messages to a given channel `cino` without waiting for a response, as follows, where each message of `size`-byte length stored in a buffer `buf`:

```
chan_send(cino, buf, size).
```

A consumer can asynchronously receive whole or prefixes of messages in the order they were sent, storing the received data into `buf`, by calling `size = chan_recv(cino, buf)`. In the following simple program fragment, the current thread creates two child threads `pch, cch` and a channel `cino`, and let the two children communicate directly via the channel.

```
spmc_init(); // initialize SPMC meta-data
int cino = chan_alloc(); // allocate a channel cino
thrd_args a={.cino = cino,...};// user-defined arguments

pch = thread_alloc(pID);//allocate a child for producing
cch = thread_alloc(cID);//allocate a child for consuming

chan_setprod(cino, pch, 0);//set pch the producer of cino
chan_setcons(cino, cch); //set cch the consumer of cino

thread_start(pch, prod, &a));//start pch to run prod(&a)
thread_start(cch, cons, &a));//start cch to run cons(&a)
```

DetMP implements each channel as a fixed-size SPMC region holding a series of messages sent by the producer. Each message occupies a contiguous range of pages in the region, and the producer fixes one or more pages for each message sent. To support asynchronous communication, DetMP maintains channel meta-data recording the status of active channels, such as IDs of the producer/consumers, and offsets at which the producers and consumers put/get the next message in the SPMC region.

### *Collective Communication.*

DetMP provides a set of collective communication functions similar to those in MPI, offering convenient high-level communication among a group of threads. Unlike MPI, collective communication functions in DetMP explicitly specify the relevant channels explicitly. A group of $n$ threads, represented as a thread pool, consists of a master thread and its $(n-1)$ child threads.

DetMP implements various MPI-like collective communication operations atop the SPMC primitive. Assuming a group with three threads, G={T0,T1,T2}, Figure 2 shows how these functions map to basic channel send/receive operations on channels.

**Barrier:** At a barrier, each DetMP child thread makes a Ret system call to stop and wait for the master. The master makes $(n-1)$ Get system calls to synchronize with each child, after which the master restarts all children to proceed past the barrier.

**Broadcast:** The root thread T0 broadcasts to all threads of a group via a *root-to-all* $(1:(n-1))$ channel c, as shown in Figure 2(a), where T0 broadcasts message 1 to T1 and T2 via channel c.

**Scatter:** The root thread T0 scatters data to all threads in a group. For $n$ threads, the function needs $(n-1)$ *root-to-thread*(1:1) channels. The root thread sends each item to its corresponding thread in turn (except the root thread itself) on the appropriate channel. In Figure 2(b), T0 sends message 2 to T1 via channel c1 and message 3 to T2 via c2, leaving message 1 for itself.

**Gather:** This function is the reverse of the *scatter*, which gathers data from all threads to the root thread. For $n$ threads, the operation needs $(n-1)$ *thread-to-root*(1:1) channels; each thread except the root sends a message to the root via the appropriate channel. In Figure 2(c), T0 receives 2 from T1 and 3 from T2 in turn via channels c1 and c2, respectively, gathering all messages in its buffer.

**Reduce:** This function combines data items sent by each thread using a specified reduction operation, such as *sum, maximum, minimum, product*, into the root thread's reduction buffer. The function

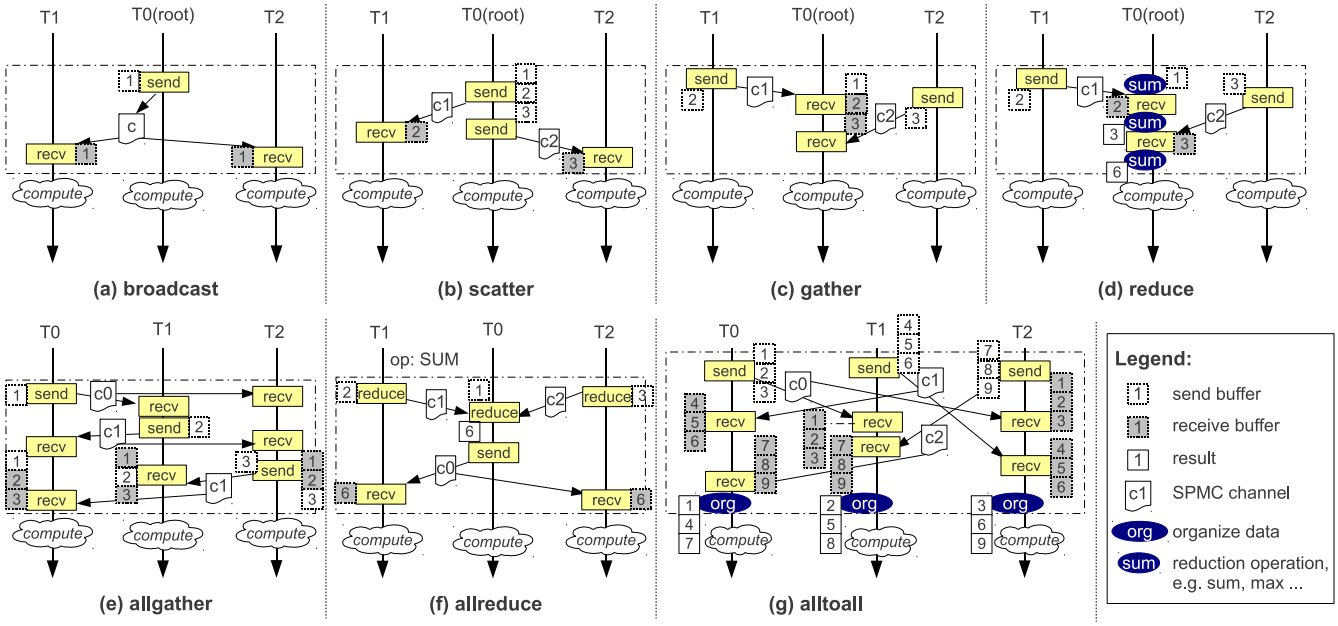


**Figure 2: Mapping of collective communication operations to SPMC channel send/recv pairs**

needs $(n-1)$ *thread-to-root*(1:1) channels for $n$ threads. In Figure 2(d), T0 first initializes the buffer as 1, then receives 2 from T1 and executes *sum* with 1, then receives 3 from T2 and executes *sum*, finally getting the reduction result 6.

**Allgather:** This function is similar to the *gather*, but all threads receive copies of all messages, not just the root thread. For $n$ threads, the function needs $n$ *thread-to-all*(1:(n-1)) channels: each thread sends a message and receives messages from other $(n-1)$ threads in the specified order. In Figure 2(e), all threads get the gathering result containing a sequence of 1, 2 and 3.

**Allreduce:** This function combines the data sent by each thread using a reduction operation, and all threads receive the result. DetMP implements this by combining *reduce* and *broadcast*: in Figure 2(f), the three threads first call *reduce* to let the root T0 get the reduced result 6, then T0 broadcasts 6 to the others via channel `c0`.

**All-to-all:** This is an extension of *allgather*, where each thread sends distinct data to each receiver. The $j-$th block sent from thread $i$ is placed in the $i-$th block of thread $j$'s receive buffer. For simplicity, as shown in Figure 2(g), each thread first broadcasts all blocks in its sending buffer to all others, and receives all other threads' blocks, then puts the relevant blocks in its receive buffer.

# 4. PROTOTYPE IMPLEMENTATION

We have extended Determinator with SPMC regions at kernel level and DetMP at user level. Though early and incomplete, the prototype is sufficient to explore the feasibility of our design.

## 4.1 Design

Determinator is written in C with small assembly fragments, currently runs on the 32-bit x86 architecture, and supports up to 1GB of physical memory directly mapped into kernel space.

We modified the Determinator kernel's virtual memory system and system calls to support SPMC regions. The kernel uses x86 page-based address translation [18] to give each process an independent user-level address space, enforcing inter-process protection and isolation. A classic x86 two-level page directory/table hierarchy maps virtual to physical addresses in 4KB pages. Each page table entry (PTE) points to physical memory page. We use an available PTE bit to record whether a mapping is to an SPMC page. The kernel sets this bit when creating an SPMC region via the Get system call, and clears this bit at a producer's Fix request.

At user level, we reserve an address range for channel meta-data and channel internal nodes or *inodes*. For simplicity each active channel inode occupies a 4MB SPMC region. To support messages with different sizes, a message in an inode consists of its real length and its content. Messages in an inode are page-aligned so as to leverage the kernel's copy-on-write optimizations. To keep meta-data consistent across processes, we separate the creation and execution of a user process as in Java, and let a parent process copy or transfer the consuming/producing right of an SPMC region to a child between the creation and execution stages of the child, as in the program fragment in Section 3.2.

We provide each process a private heap for dynamic memory allocation (`malloc`) in applications.

## 4.2 Preliminary Results

We first compared DetMP with Determinator's original shared memory programming model (DetSM) using **matmult (MM)**, which multiplies 1024-by-1024 matrices. Both implementations divide work in the same way and use a master/slave coordination model. In MM-DetMP, the master uses channels to communicate with each child sending the task and receiving the result. We divide total execution time into *computation* (CP), *communication* (CM), and *verification*. In MM-DetSM, the master uses *copy* at the *fork* stage to transfer a task to a child, and *merge* at the *join* stage to collect the result from a child. We split total running time into *computation*, *merging* (MG), and *verification*. We omit verification time as it represents a negligible percentage of the total.

Figure 3(a) shows each version's speedup (SP) relative to single-CPU execution on the extended Determinator ran under QEMU [6] on a 48 core, 1.7GHz AMD Opteron PC. Table 2(a) lists the specific statistics, including time percentages of computation, communication and merging occupied by each thread on average, and the specific speedup and speedup ratio. MM-DetMP scales better than

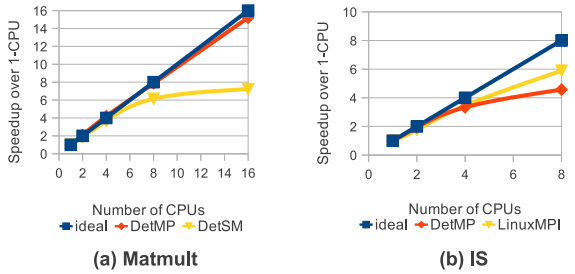


**Figure 3: Parallel speedup over its own single-CPU performance on various benchmarks.**

**Table 2: Scalability and time distribution comparison.**

**(a) MM-DetMP vs. MM-DetSM.**

| CPUs | DetMP | | | DetSM | | | DetMP-SP/ |
|---|---|---|---|---|---|---|---|
| | CP | CM | SP | CP | MG | SP | DetSM-SP |
| 1 | 99.9% | 0.0% | 1 | 100.0% | 0% | 1 | 1 |
| 2 | 99.6% | 0.3% | 2.09 | 98.7% | 1.3% | 1.89 | 1.11 |
| 4 | 99.2% | 0.5% | 4.19 | 97.0% | 2.9% | 3.67 | 1.14 |
| 8 | 98.9% | 0.8% | 7.86 | 93.6% | 6.4% | 6.14 | 1.28 |
| 16 | 98.4% | 1.3% | 15.21 | 87.5% | 12.5% | 7.23 | 2.1 |

**(b)IS-DetMP vs. IS-LinuxMPI.**

| CPUs | DetMP | | | LinuxMPI | | | DetMP-SP/ |
|---|---|---|---|---|---|---|---|
| | CP | CM | SP | CP | CM | SP | LinuxMPI-SP |
| 1 | 92.3% | 7.5% | 1 | 92.6% | 7.3% | 1 | 1 |
| 2 | 73.0% | 26.8% | 1.93 | 41.6% | 57.6% | 1.75 | 1.1 |
| 4 | 67.6% | 32.0% | 3.33 | 30.1% | 69.5% | 3.45 | 0.97 |
| 8 | 48.3% | 51.6% | 4.57 | 24.2% | 75.3% | 5.89 | 0.78 |

MM-DetSM, and achieves close to ideal scalability. The main reason is that merging in MM-DetSM occupies more time percentage than communication in MM-DetMP with increasing thread count, as shown in Table 2(a).

We also compared DetMP with Linux MPI by porting the Integer Sort (IS) benchmark from NPB3.3-MPI [12] to DetMP. IS is a bucket sort, where the number of keys ranked, number of processors used, and number of buckets employed are all powers of two. Communication costs are dominated by an *Alltoallv* operation.

Figure 3(b) shows the speedup of IS-DetMP running on the extended Determinator, and IS-LinuxMPI running on Linux, relative to the single-CPU performance of each. Table 2(b) lists specific measurements of time distribution, speedup and speedup ratio. IS-LinuxMPI ran on Ubuntu 10.10 with MPICH2, and both Ubuntu and Determinator were run in the same QEMU environment on the above 48-core PC. We show only results with up to 8 CPUs because when running under QEMU, Ubuntu uses at most 8 CPUs regardless of the number of CPUs specified to QEMU. Speedup of IS-DetMP is lower than that of IS-LinuxMPI on 4 and 8 CPUs (which are 97% and 78% of IS-LinuxMPI's, respectively), but it still scales well and, unlike Linux, guarantees determinism. We also see that the communication in IS-DetMP takes a smaller time percentage than in IS-LinuxMPI with an increasing number of thread.

# 5. CONCLUSION

We believe that SPMC offers a promising virtual memory foundation for implementing scalable, deterministic, and convenient parallel programming models, for today's and tomorrow's massively multicore processors. The kernel's SPMC primitive offers non-hierarchical inter-process communication and synchronization while ensuring system-enforced determinism. DetMP has shown good scalability on the benchmarks evaluated so far. In the future we expect to explore more efficient channels and other parallel programming models atop SPMC.

*Acknowledgments.*

This research was supported by China's Fundamental Research Funds for the Central Universities, and by the U.S. National Science Foundation under grant CNS-1017206.

# 6. REFERENCES

[1] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *VVEIS*, pages 82–93, April 2003.

[2] Amittai Aviram et al. Determinating timing channels in compute clouds. In *CCSW*, October 2010.

[3] Amittai Aviram and Bryan Ford. Deterministic OpenMP for race-free parallelism. In *3rd HotPar*, May 2011.

[4] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A programming model for shared memory parallelism. In *2nd WoDet*, March 2011.

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Determinator: OS support for efficient deterministic parallelism. In *9th OSDI*, October 2010.

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator, April 2005.

[7] Tom Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, March 2010.

[8] Emery D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, October 2009.

[9] Robert L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, October 2009.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd OSDI*, pages 173–186, February 1999.

[11] Heming Cui, Jingyue Wu, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *9th OSDI*, October 2010.

[12] Rob F. Van der Wijngaart. NAS parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, October 2002.

[13] Joseph Devietti et al. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, March 2009.

[14] George W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th OSDI*, December 2002.

[15] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *Transactions on VLSI Systems*, 14(8):854–867, August 2006.

[16] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, March 2008.

[17] Bryan Ford et al. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[18] Intel Corporation. IA-32 Intel architecture software developer's manual, June 2005.

[19] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, Amsterdam, Netherlands, 1974. North-Holland.

[20] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, pages 1–15, April 2005.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard version 2.2, September 2009.

[22] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *14th ASPLOS*, March 2009.

[23] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Scaling deterministic multithreading. In *2nd WoDet*, March 2011.

[24] Angela C. Sodan. Message-passing and shared-data programming models – wish vs. reality. In *19th HPCA*, May 2005.

# Static Analysis of Device Drivers: We Can Do Better!

Sidney Amani[‡§] Leonid Ryzhyk[‡§] Alastair F. Donaldson[¶]
Gernot Heiser[‡§] Alexander Legg[‡‖] Yanjin Zhu[‡§]
[‡]NICTA* [§]University of New South Wales [¶]University of Oxford† [‖]University of Sydney
sidney.amani@nicta.com.au

## ABSTRACT

We argue that the device driver architecture enforced by current operating systems complicates both manual and automatic reasoning about driver behaviour. In particular, it makes it hard and in some cases impossible to statically verify that the driver correctly interacts with the rest of the kernel. This limitation cannot be addressed solely via better verification tools. We maintain that qualitative improvement in the effectiveness of static driver verification must rely on an improved driver architecture, leading to drivers that are easier to write, understand, and verify.

To support our claims, we present a device driver architecture, called active drivers, that satisfies these requirements. We outline our methodology for specifying and verifying active driver protocols using existing model checking tools and describe initial experimental results.

## Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Input/Output; B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices

## General Terms

Reliability, Verification

## Keywords

Device Drivers, Software Protocols, Model Checking

## 1. INTRODUCTION

Faulty device drivers are a major source of operating system (OS) failures. In this paper we focus on a specific class of driver bugs—violations of protocols between the driver and the rest of the OS. These bugs are becoming increasingly common due to the growing complexity of internal OS interfaces [1]. Our earlier study [9] showed that OS protocol violations account for 20% of all driver bugs.

Automatic verification tools have proved useful in detecting driver protocol violations. In fact, some of the most advanced verification tools for C, including SLAM [1], SATABS [3], and Blast [7] were either designed with the purpose of verifying driver protocols or used driver verification as their primary case study.

Despite significant effort invested in improving these tools, they remain limited in the size and complexity of drivers that they can handle and properties that they can verify for these drivers without generating a large number of false positives.

We argue that to a large extent these limitations are due to the device driver architecture enforced by current operating systems. In this architecture the driver does not have its own thread of control and instead its functions are called directly by OS threads. If the driver makes incorrect assumptions about the order in which the OS can invoke its entry points, it will be unable to handle these invocations correctly. As these assumptions are implicit in the source code of the driver, such bugs can only be detected via their indirect consequences, which can be difficult or impossible to identify automatically.

The following code fragment, showing two driver entry points, illustrates the problem:

```
int suspend() {... free (p); ...}
void remove() {... p->data=0; ...}
```

This code assumes that the `remove()` entry point cannot be called after `suspend()` and therefore it is safe to deallocate pointer `p` inside `suspend()`. While uncommon, this sequence of driver invocations can occur in practice if the user unplugs the device while it is in the suspended state, leading to an invalid pointer dereference.

The problem in this example is that the root cause of the bug—the fact that the driver is not prepared to handle `remove()` after `suspend()`—is implicit in the driver code and cannot be discovered using control flow analysis. The model checker can still find the bug by employing a hand-written C model of the OS kernel that randomly generates all possible sequences of driver invocations by the OS [1] and detecting that one of these sequences leads to a use-after-free error; however such analysis quickly gets intractable for code involving complex pointer manipulation. For instance, if the `remove()` function accessed the content of `p` via an alias pointer, this could lead to very long verification time.

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

†Alastair F. Donaldson is supported by EPSRC grant EP/G051100.

Furthermore, if the problematic sequence of invocations caused the driver to issue invalid commands to the device instead of performing an invalid pointer dereference, such a bug would even in principle be impossible to detect automatically without providing a formal model of the device to the model checker.

We believe that qualitative improvement in the effectiveness of automatic driver verification must rely on an improved driver architecture. In this architecture, interactions with the OS must be explicit in the code of the driver, which allows verifying the driver's compliance with OS protocols using control flow analysis.

In our earlier work [10], we proposed such an architecture, called *active drivers*. It was originally introduced with the goal of improving driver reliability by making drivers easier to write and understand.

In the present paper we show that the active driver architecture also facilitates automatic analysis of driver correctness. To this end, we specify interaction protocols between active drivers and the OS using finite state machines. The driver implementation is then automatically checked for compliance with the protocol state machine using an existing model checker, namely SATABS. Our initial experiments show that this methodology allows verification of driver properties that are hard or impossible to check for conventional drivers.

In the rest of the paper we give an outline of the active driver architecture (Section 2), present our methodology for writing formal specifications of active driver protocols (Section 3), and describe how these protocols can be verified using an existing C model checker (Section 4). We summarise our experimental results in Section 5 and discuss related work in Section 6.

## 2. ACTIVE DRIVERS

Active device drivers [10] were introduced to address two issues with the conventional driver architecture, where a driver is a passive object that is only activated when an external thread invokes one of its entry points. First, since multiple threads can invoke the driver concurrently, it must take care to synchronise the invocations to avoid race conditions. Second, since the driver does not have its own thread of control, it cannot rely on programming-language constructs to maintain its control flow and instead records its execution state using state variables. Both issues make the logic of the driver difficult to implement and even harder to understand and modify.

In the active driver architecture, every driver runs in the context of its own thread. Communication between the driver thread and other OS threads occurs via message passing. The driver-OS interface consists of a set of *mailboxes* where each mailbox is used for a particular type of message. The OS sends I/O requests and interrupt notifications to the driver by placing them in appropriate mailboxes. The driver receives a message by performing a blocking wait on one or more mailboxes. The driver notifies the OS about a completed request via a reply mailbox. A message can carry a payload consisting of one or more arguments. The number and types of arguments is determined by the message type.

An active driver is structured as a sequential program, with the order in which the driver handles OS requests explicitly defined in the structure of the program. In addition, since the driver handles all requests in the context of its own thread, it does not have to worry about thread synchronisation.

An active driver can register several interfaces with the OS. For each interface, the OS creates a set of mailboxes for communication with the driver.

Support for active drivers can be added to an existing OS kernel. To this end, the OS must be extended with interface adapters that perform the conversion between native driver interfaces supported by the OS and message-based interfaces implemented by the driver.

The driver exchanges messages with the OS via `EMIT` and `AWAIT` primitives. The `EMIT` function places a message in a mailbox. The sending thread continues without blocking. The `AWAIT` function takes references to one or more mailboxes. If there is a message queued at one of these mailboxes, `AWAIT` returns immediately. Otherwise, it blocks until a message arrives to one of the mailboxes. In either case, it returns a reference to the mailbox containing the message. A mailbox can queue multiple messages. `AWAIT` always dequeues exactly one message, which is accessible via a pointer in the returned mailbox.

We illustrate how the active driver architecture facilitates static analysis of device drivers using a fragment of active driver code that matches the example from Section 1:

```
1 mb = AWAIT(suspend, remove, ...);
2 if (mb == suspend) {
3     ...
4     free(p);
5     mb = AWAIT(resume);
6     ...
7 } else if (mb == remove) {
8     p->data = 0; ...
9 }
```

Here, `suspend`, `resume`, and `remove` are pointers to driver mailboxes. In line 1 the driver waits for both suspend and remove requests. After receiving a suspend request in line 2, the driver deallocates pointer `p` and waits for a resume request in line 5.

This implementation has an equivalent bug to the one found in the original, passive, version of the driver: the driver does not handle remove requests in the suspended state. A correct implementation must wait on both `resume` and `remove` mailboxes in line 5. Otherwise the driver deadlocks if `remove` rather that `resume` arrives while it is blocked at line 5.

The key difference between the passive and the active versions of the driver is that the active version better captures the programmer's intention and achieves cleaner separation of control and data flows. In this version, the bug is evident in the control flow of the driver and can be discovered without resorting to pointer analysis. Upon discovering the bug, the driver developer adds the `remove` message to the `AWAIT` call in line 5 and implements code to handle this message. It is clear from the control flow of the driver that this code is invoked in the suspended state and therefore it is likely to differ from the `remove` handler for the full-power state in line 8.

As can be seen from the above example, active driver code is more verbose compared to passive drivers. This results in a small increase in the code size for a complete driver (see Section 5). We believe that this overhead is justified by the improved clarity of active drivers.

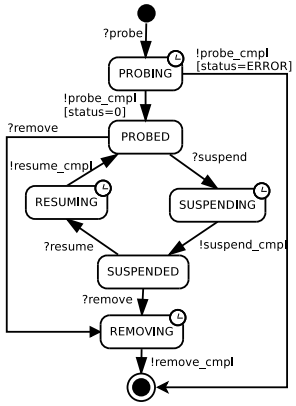


**Figure 1: PCI protocol fragment.**

## 3. SPECIFYING DRIVER PROTOCOLS

In order to enable formal analysis of the driver-OS interaction, we associate a behavioural protocol with each active driver interface. The protocol specifies messages from the OS that the driver must be prepared to handle in every state as well as messages that the driver is allowed to send to the OS. Driver protocols are defined by the I/O framework designer and are generic in the sense that every driver that implements the given interface must comply with the associated protocol.

We specify driver protocols in the form of finite state machines (FSMs) where every state transition corresponds to a message sent or received by the driver. The protocol state machine conceptually runs in parallel with the driver: whenever the driver sends or receives a message that belongs to the given protocol, this triggers a matching state transition in the protocol state machine.

The use of FSMs to formalise driver behaviour was proposed in the Dingo [9] driver framework. While Dingo aims to make driver protocols easier to understand and to check at runtime, our focus is on static verification. Therefore we use a subset of the Dingo specification language that lends itself naturally to static analysis.

Figure 1 shows a simplified version of the protocol that must be implemented by any driver for a PCI-based device. This protocol describes the handling of initialisation, shutdown, and power management requests by the driver. Each protocol state transition is labelled by the name of the mailbox through which the driver sends or receives a message, with exclamation marks ("!") denoting send transitions and question marks ("?") denoting receive transitions. The label can also contain an optional guard condition in square brackets, which constrains values of message arguments.

According to Figure 1, in the initial state the driver must wait for a `probe` message from the OS. In response to this message, the driver initialises the device and notifies the OS about the completion of this operation via a `probe_cmpl` message. If initialisation is successful (the `status` argument of `probe_cmpl` message is 0), the protocol state machine moves to the `PROBED` state; otherwise it reaches its final state. The remaining protocol state transitions in Figure 1 can be interpreted analogously.

In some states of the protocol the OS is waiting for the driver to complete a request. The driver cannot remain in such a state indefinitely, but must eventually leave the state by sending a response message to the OS. Such states are called *timed* states and are labelled with the clock symbol in Figure 1.

We use the Statecharts [6] syntax to achieve compact representation of complex protocol state machines. Statecharts organise states into a hierarchy, where several primitive states can be clustered into a super-state. Super-states can be composed sequentially or in parallel. The latter is useful for specifying several independent activities within a protocol, e.g. sending and receiving of network packets.

In order to enable automatic verification of drivers' protocol compliance, we define a set of rules that must be followed by any driver that implements a protocol:

1. *EMIT:* The driver is allowed to emit a message to a mailbox if and only if this message triggers a valid state transition in the protocol state machine.
2. *AWAIT:* Whenever the driver performs an `AWAIT` operation, it must wait on all incoming mailboxes enabled in the current state of the protocol.
3. *Timed:* The driver must not remain in a timed state forever.
4. *Termination:* When the main driver function returns, the protocol state machine must be in a final state.

The first rule forbids the driver to send a message that the OS does not expect in the current state. The second rule prevents the driver from waiting for only a subset of enabled incoming messages. Violation of this rule can lead to a deadlock, as shown in Section 2. The third rule forces the driver to respond to all OS requests by eventually exiting each timed state of the protocol. Finally, the fourth rule makes sure that the driver does not terminate leaving some of its protocols in intermediate states.

Rule 2 is too restrictive in practice. It requires the driver to wait for all enabled messages of all its protocols. As a result, the driver is unable to deliberately delay handling of some of the messages, which is useful for instance if the driver wants to complete the current request before checking for more requests from the OS. In order to enable delayed message handling without sacrificing correctness, we use a relaxed version of rule 2 that allows the driver to wait for a subset of enabled protocol messages as long as one of these messages is guaranteed to be received eventually in the current state. The protocol designer must mark such safe sets of messages in the protocol state machine.

## 4. VERIFYING DRIVER PROTOCOLS

In order to automatically verify driver protocols, we must first convert them into a format understood by verification tools. Most verification tools for C handle properties expressed as source code assertions. Therefore we encode rules 1 and 2 into assertions embedded in modified versions of `EMIT` and `AWAIT` used for verification. These modified functions keep track of the state of the protocol and check that each `EMIT` and `AWAIT` call issued by the driver complies with rules 1 and 2. At the moment, these checks must be written manually based on protocol specifications. Automating this task is part of ongoing work.

In order to verify rule 4, we implement a wrapper function around the driver's main function, which checks that upon return from main the protocol state machine is in one of its final states.

The above approach to driver verification does not require modifications to driver source code. Furthermore, it verifies drivers compositionally, one protocol at a time.

The main limitation of assertion-based verification is that it does not handle liveness properties, i.e. properties whose violation cannot be demonstrated using a finite trace of the program. Rule 3 falls into this category. It requires the driver to eventually respond to any OS request. This rule cannot be violated in any finite sequence of steps; however an infinite run of the driver can violate it.

Liveness properties can be formalised using temporal logic and verified with the help of a temporal logic model checker [5]. Results presented in this paper were obtained using the SATABS model checker, which only supports assertion-based verification. Therefore we did not verify rule 3 in our experiments. Further work on the project will address this limitation.

**Speeding up verification** We have experimented with active driver protocol verification using SATABS [3], a counterexample-guided abstraction refinement (CEGAR) model checker for C.

Although manual analysis of driver code confirms that the control structure of active drivers is largely independent of the data path and therefore can in principle be verified efficiently, our initial experiments showed poor verification performance, with most verification runs not finishing within reasonable time. Analysis of performance issues led to an improved encoding of protocol state machines and tuning of internal SATABS heuristics.

First, we discovered that verifying a complete protocol in one go leads to overly complex abstractions. As the model checker tries to verify various assertions that encode the protocol, it adds new predicates to the abstract model of the driver. While the number of predicates involved in verifying an individual protocol state transition is typically small, the overall number of predicates generated with this approach overwhelms the model checker.

We address the problem by decomposing each driver protocol state machine into a set of much simpler protocols. The decomposition is equivalent to the original protocol, i.e. the driver satisfies the original protocol if and only if it satisfies each protocol in the decomposition. In our experience, even complex driver protocols allow decomposition into simple protocols with no more than 4 states and only a few transitions.

At the moment, we perform the decomposition manually using the following heuristic rule: every protocol in the decomposition must capture a single constraint on the driver behaviour. One example of such a rule that can be extracted from the PCI protocol (Figure 1) is that after sending a `probe_complete` message the driver must be prepared to handle `remove` and `suspend` messages from the OS. Protocol decomposition only needs to be performed once for a class of device drivers. Nevertheless it is a tedious and error-prone task; therefore we plan to automate it in the future.

Verifying each protocol in the decomposition requires only a small subset of predicates involved in checking the monolithic protocol, leading to exponentially faster verification. By checking each individual protocol in parallel, the monolithic protocol can be verified in the time required by the slowest protocol in the decomposition.

Second, we found that in analysing a spurious counterexample, predicates extracted from program points close to the failed assertion tended to be useful in allowing verification to progress, while predicates less close to the error were often irrelevant to the property being checked. Thus, we added to SATABS a feature to introduce no more than a user-specified number of new predicates (usually between 1 and 4) during each CEGAR iteration, favouring predicates derived from program points close to the failed assertion.

Third, we noticed that by default SATABS introduced a large number of pointer-related predicates. Most of these predicates do not affect driver's control flow and are therefore irrelevant to verifying driver protocols. Thus we added to SATABS a heuristic to favour non-pointer predicates, that is predicates which are not equality tests between pointer expressions, when adding new predicates.

# 5. EVALUATION

In order to evaluate the active driver architecture and its impact on driver verification, we specified and implemented protocols for several classes of device drivers, including Ethernet, SCSI, and ATA driver protocols. We based our protocol specifications on equivalent Linux driver interfaces, to simplify porting of existing Linux drivers to the active architecture. The drivers we ported are the RTL8169 Ethernet controller driver, the generic SCSI-to-ATA converter driver (`libata`), and the AHCI SATA controller driver.

Our initial verification results are based on the AHCI controller driver. We chose this driver due to the extreme complexity of its protocol. The Linux ATA protocol that this driver implements emerged as a result of splitting a monolithic ATA controller driver into a device-independent layer (`libata`) and device-specific drivers for various ATA controllers. The two layers interact via numerous nested callbacks most of which can only be invoked in a specific context. Our specification of the ATA protocol state machine consists of 39 states and 61 transitions. We decomposed this protocol into an equivalent set of 22 protocols, each consisting of 2 to 4 states. The native Linux AHCI driver contains 2268 lines of C code; the equivalent active driver is 2427 loc.

Using the modified version of SATABS described in the previous section, we were able to verify all safety properties of ATA and PCI protocols. The longest checker took 12 hours.

In all cases, analysis of SATABS output revealed that most predicates required to verify driver protocols were control-flow related, with only a small number of data-related predicates. This shows that the active driver architecture effectively separates the control logic of the driver responsible for interaction with the OS from its data flow, which enables efficient verification of driver protocols.

We evaluate the performance overhead of active drivers by comparing the performance of the native Linux driver and the active driver for the AHCI controller using the `iozone` benchmark suite running on a system with a 2.33GHz Intel Core 2 Duo CPU with one enabled core, Marvell 88SE9123 PCIe 2.0 SATA controller, and WD Caviar SATA-II 7200 RPM hard disk. While both drivers achieved the same I/O throughput on all tests, the active driver's CPU utilisation (measured with oprofile) was slightly higher (Figure 2). This overhead is due to the higher cost of message-based communication compared to direct function calls used by the native driver. It can be reduced using an optimised implementation of the message passing mechanism and improved protocol design. Our ATA driver protocol, based on the equivalent

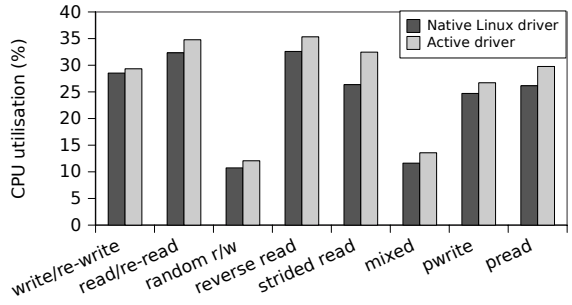


**Figure 2: Native vs. active driver performance on the iozone benchmark.**

Linux interface, requires 10 messages for each I/O operation. A clean-slate redesign of this protocol would involve much fewer messages.

## 6. RELATED WORK

Several previous systems, including Singularity [4], RMoX [2], and Dingo [9], implement variations of the active device driver architecture. Singularity and RMoX also support static verification of driver protocols. To this end, they rely on specialised language features and radically different OS architectures.

Active drivers were introduced as an OS and language-independent concept in [10], with the goal of simplifying driver development and avoiding common types of driver bugs. In the present work we demonstrate that active drivers also facilitate static verification of driver-OS protocols for drivers written in C and for conventional OSes.

Tools like SLAM [1], SATABS [11], and Coccinelle [8] have been used for static analysis of Windows and Linux device drivers. As discussed in Section 1, the power of these tools is limited by the conventional driver architecture. In this work we demonstrate that the same tools can be used to verify complete driver-OS protocols for active drivers.

## 7. CONCLUSION

We argued that the effectiveness of automatic verification tools in finding driver bugs can be increased with the help of an improved device driver architecture where interactions with the OS are explicit in the source code of the driver. We presented such an architecture and showed that it enables the use of existing model checking tools to verify properties that are hard or impossible to check on conventional drivers.

## Acknowledgement

We are grateful to Michael Tautschnig for assistance with the use of SATABS.

## 8. REFERENCES

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *1st EuroSys Conf.*, pages 73–85, Leuven, Belgium, Apr 2006.

[2] F. Barnes and C. Ritson. Checking process-oriented operating system behaviour using CSP and refinement. *Operat. Syst. Rev.*, 43(4):45–49, Oct 2009.

[3] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[4] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *1st EuroSys Conf.*, pages 177–190, Leuven, Belgium, Apr 2006.

[5] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna — A Static Model Checker. In *11th FMICS*, pages 297–300, Bonn, Germany, Aug 2006.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, Jun 1987.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *14th Int. Conf. Comp. Aided Verification*, pages 526–538, Copenhagen, Denmark, 2002.

[8] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: ten years later. In *16th ASPLOS*, pages 305–318, Newport Beach, CA, USA, 2011.

[9] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *4th EuroSys Conf.*, Nuremberg, Germany, Apr 2009.

[10] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *1st APSys*, pages 25–30, New Delhi, India, Aug 2010.

[11] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *22nd ASE*, pages 501–504, Atlanta, Georgia, USA, 2007.

# Retroactive Auditing

Xi Wang Nickolai Zeldovich M. Frans Kaashoek
MIT CSAIL

## ABSTRACT

Retroactive auditing is a new approach for detecting past intrusions and vulnerability exploits based on security patches. It works by spawning two copies of the code that was patched, one with and one without the patch, and running both of them on the same inputs observed during the system's original execution. If the resulting outputs differ, an alarm is raised, since the input may have triggered the patched vulnerability. Unlike prior tools, retroactive auditing does not require developers to write predicates for each vulnerability.

## 1. INTRODUCTION

Due to the increasing size and complexity of software, software defects have become inevitable [2]. These defects often lead to vulnerabilities, and allow an adversary to break into the system, until developers release updates that fix the defects and administrators patch the system with those updates. Once the administrators install an update, they may want to know whether some adversary already exploited the corresponding vulnerability before the patch was installed.

This position paper proposes a new tool, RAD, to help administrators audit the past execution of their system and detect intrusions. RAD's workflow is shown in Figure 1. After a patch is released, RAD re-executes all programs that invoked the vulnerable code, both with and without the patch applied, and feeds in the inputs seen during the system's original execution. If the programs behave differently, an adversary may have exploited this vulnerability. As we describe in Section 6, unlike previous approaches [5], RAD does not require developers to write predicates for each vulnerability.

The key challenge facing RAD is reducing false alarms. Executing the patched code may lead to a different bit-level system state, even if that state is functionally identical to the original one. For example, the program may be non-deterministic, in which case each run would produce a different result. A program might also create a complex data structure, such as a binary tree, which may end up having different pointer values, due to differences in `malloc`'s behavior, even though the trees are equivalent.

The main contribution of this paper is the idea of retroactive auditing. Based on a visual inspection of past vulnerabilities discovered in the Apache web server, we argue that this approach should work well in practice. We further propose several ideas for reducing false alarms.



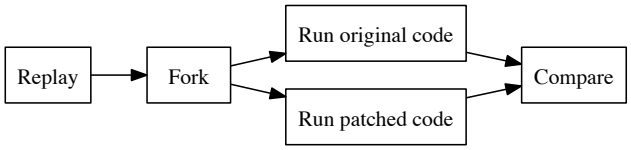


**Figure 1: The workflow of retroactive auditing.**

The rest of the paper is organized as follows. Section 2 gives an overview of RAD. Section 3 presents a feasibility study using vulnerabilities from Apache, based on visual inspection. Section 4 shows some initial results for applying RAD to two vulnerabilities. Section 5 discusses future challenges. Section 6 relates RAD to previous work and Section 7 concludes.

## 2. OVERVIEW

This section describes how RAD works, using CVE-2009-0023 shown in Figure 2 as a running example. This vulnerability was discovered in the APR-util library bundled with the Apache web server. At line 7, the developers mistakenly used a signed integer `(int)s[i]` as an index of array `shift`; the index could be negative, which allows an adversary to craft an input string to overwrite heap memory. Line 14 contains a similar bug. Patching the problem is straightforward: use an unsigned index. RAD's goal is to apply the patch retroactively to a system and determine if an attacker used the vulnerability to compromise the system.

### 2.1 A strawman approach

Consider a simple command-line text search program `xgrep`, which reads from stdin and prints results to stdout, just like standard `grep`, except that it makes use of the vulnerable function in our running example (see Figure 2). We assume that the administrator has deployed a monitoring infrastructure that records the past execution of the system and is able to replay the system's execution from a previous state (e.g., using Retro [7]). To determine whether any past execution of `xgrep` was exploited, a strawman version of RAD works as follows.

First, RAD rolls the system back to a state before the exploit could have occurred (e.g., when the vulnerable software was first installed).[1] Then, RAD replays the system's execution using the inputs recorded during the original run. During replay, before each execution of `xgrep`, RAD spawns both a copy of `xgrep` using the

[1]Administrators may need to make trade-offs between a longer time horizon for auditing and higher storage costs for audit logs.

```
--- apr/apr-util/branches/1.3.x/strmatch/apr_strmatch.c ...
+++ apr/apr-util/branches/1.3.x/strmatch/apr_strmatch.c ...
@@ -103,12 +103,13 @@
 if (case_sensitive) {
   pattern->compare = match_boyer_moore_horspool;
   for (i = 0; i < pattern->length - 1; i++) {
-    shift[(int)s[i]] = pattern->length - i - 1;
+    shift[(unsigned char)s[i]] = pattern->length - i - 1;
   }
 }
 else {
   pattern->compare = match_boyer_moore_horspool_nocase;
   for (i = 0; i < pattern->length - 1; i++) {
-    shift[apr_tolower(s[i])] = pattern->length - i - 1;
+    shift[(unsigned char)apr_tolower(s[i])]
+      = pattern->length - i - 1;
   }
 }
```

**Figure 2: The patch for CVE-2009-0023, heap underwrites in APR-util function `apr_strmatch_precompile`.**

original, vulnerable APR-util library, and a copy of `xgrep` using the patched APR-util library. RAD then runs the two processes, collects the outputs from both runs, and compares their outputs. If there is no difference, RAD concludes that the vulnerable function was not exploited; otherwise RAD reports a possible exploit. This conclusion assumes that the patch was designed to address the vulnerability and that the patch addresses the vulnerability correctly.

The strawman approach re-executes any process, in its entirety, that may have been affected by the patched code. This poses several challenges when applying the same approach to a complex program like the Apache web server. First, it is costly to re-execute the entire process, and it is often unnecessary, because a patch is often small [10] and typically causes small changes in a program's behavior. Second, programs such as Apache often involve several threads or processes, and their behavior can be non-deterministic even with the same inputs. As a result, it is difficult to match two non-deterministic runs, even if the patch does not functionally change anything.

## 2.2 Fine-grained auditing

RAD addresses the above challenges by using fine-grained re-execution and auditing. Instead of auditing the whole process, RAD can audit runs of a smaller code piece that is likely to be deterministic (e.g., a single function in Figure 2) as follows.

Again, RAD rolls back the system to a correct state, and replays the past executions of vulnerable processes. Since the patch modifies only one function, `apr_strmatch_precompile`, RAD intercepts every call to that function in all processes (currently, it uses `LD_PRELOAD` on Linux to inject a code stub).

Before every invocation of the vulnerable function, the injected code stub forks the calling process into two: one fork invokes the vulnerable function and the other fork invokes the patched function. RAD runs the two versions of the function and records the memory writes they perform during their execution. This recording is implemented using Pin [8], which can instrument load and store instructions.

RAD compares the resulting system state of the two processes, by doing a diff of the recorded memory writes. In our example, if an adversary did not compromise the vulnerable function, then the stores to the array `shift` are within its boundary. Thus both versions of the function will behave in the same way (e.g., there will be no difference in the recorded writes), and RAD will report no warnings. If an adversary enticed the program to use a negative index, however, the underwrites of `shift` will happen in the vulnerable code, but not in the patched code. In this case the diff between the recorded

```
--- modules/proxy/mod_proxy_ftp.c ...
+++ modules/proxy/mod_proxy_ftp.c ...
@@ -385,4 +385,5 @@
 if (wildcard != NULL) {
+  wildcard = ap_escape_html(p, wildcard);
   APR_BRIGADE_INSERT_TAIL(out, apr_bucket_pool_create(
     wildcard, strlen(wildcard), p,
     c->bucket_alloc));
```

**Figure 4: The patch for CVE-2008-2939, cross-site scripting.**

memory writes will be non-empty, and RAD will report an exploit and output the diff (see Section 4.1).

This fine-grained auditing scheme imposes additional requirements on patches. Particularly, a patch should not change the function signature, such as adding or removing parameters, nor should it change the layout of any external data structures used by the function. Otherwise, the patched code cannot run starting with the same inputs from the state after fork. For patches that don't meet these requirements, RAD will fall back to the strawman approach and audit the whole program. As we will detail in Section 3, we inspected 36 patches for Apache and only 2 do not satisfy these requirements.

# 3. FEASIBILITY STUDY

This section presents a feasibility study of RAD's approach based on a visual inspection of reported vulnerabilities and the corresponding patches for the Apache web server. Figure 3 summarizes all 36 vulnerabilities announced in every Apache 2.2.x release [1], from 2005 to 2010. These vulnerabilities are located in Apache's core code, modules, lower-level libraries (APR and APR-util), and third-party libraries (expat). We determine whether RAD would be able to detect these vulnerabilities correctly or report false alarms by manually inspecting their patches.

12 vulnerabilities (marked as ∅) do not require auditing because they don't result in a compromise but just cause denial of service (i.e., crash or hang). Particularly, 5 may lead to null pointer dereference, 4 may consume more memory than necessary, 2 may hang the server, and 1 may cause the "billion laughs" attack [4], i.e., infinite recursion. None of these vulnerabilities allow attackers to break the integrity of the system or sniff any sensitive information.

Out of the 24 other vulnerabilities, we believe RAD would catch exploits and incur no false alarms for 15 of them (marked as ✓). Their patches meet the requirements discussed in Section 2.2. Section 4 uses two of them for a detailed case study.

For the remaining 9 vulnerabilities, RAD may report false alarms (marked as ☒). We further break them down into the following categories.

*Memory layout* (3). Figure 4 shows one of the cases, CVE-2008-2939. The patch allocates a new memory block to hold the sanitized input, thus the memory layouts diverge in the two executions, though they are structurally equivalent. RAD's naïve diff of writes would report false alarms for inputs that don't contain any dangerous characters.

*Character encoding* (3). The browser may be tricked into interpreting the HTTP response using an incorrect character encoding (e.g., UTF-7), if the server does not set it explicitly. The attacker can exploit this vulnerability using carefully chosen inputs to mount a cross-site scripting (XSS) attack. The patch enforces the encoding at the server side, and results in a change of every response, even where there is no compromise. Thus, RAD would report false alarms.

*Web page* (1). An administration web page provided by one of Apache's modules is vulnerable to cross-site request forgery (CSRF) attacks. The patch adds a token to the forms in the web page, which

| Version | Identifier | Component | Type | Detectability |
|---|---|---|---|---|
| 2.2.17 | CVE-2009-3720 | expat | buffer overread | ✓ |
| | CVE-2009-3560 | expat | buffer overread | ✓ |
| | CVE-2010-1623 | APR-util | resource exhaustion | ∅ |
| 2.2.16 | CVE-2010-2068 | `mod_proxy_http` | logic error | ✓ |
| | CVE-2010-1452 | `mod_cache` & `mod_dav` | null dereference | ∅ |
| 2.2.15 | CVE-2010-0425 | `mod_isapi` | logic error | ✓ |
| | CVE-2010-0434 | `mod_headers` | dangling pointers | ✓ |
| | CVE-2010-0408 | `mod_proxy_ajp` | hang | ∅ |
| 2.2.14 | CVE-2009-3094 | `mod_proxy_ftp` | null dereference | ∅ |
| | CVE-2009-3095 | `mod_proxy_ftp` | missing checks | ✓ |
| | CVE-2009-2699 | APR | hang | ∅ |
| 2.2.13 | CVE-2009-2412 | APR | integer overflow | ✓ |
| 2.2.12 | CVE-2009-1890 | `mod_proxy` | resource exhaustion | ∅ |
| | CVE-2009-1191 | `mod_proxy_ajp` | logic error | ✓ |
| | CVE-2009-1891 | `mod_deflate` | resource exhaustion | ∅ |
| | CVE-2009-1195 | config | logic error | ☒ design |
| | CVE-2009-1956 | APR-util | off-by-one | ✓ |
| | CVE-2009-1955 | APR-util | billion laughs | ∅ |
| | CVE-2009-0023 | APR-util | heap underwrite | ✓ |
| 2.2.10 | CVE-2010-2791 | `mod_proxy_http` | logic error | ✓ |
| | CVE-2008-2939 | `mod_proxy_ftp` | cross-site scripting | ☒ memory |
| 2.2.9 | CVE-2007-6420 | `mod_proxy_balancer` | cross-site request forgery | ☒ web page |
| | CVE-2008-2364 | `mod_proxy_http` | resource exhaustion | ∅ |
| 2.2.8 | CVE-2008-0005 | `mod_proxy_ftp` | cross-site scripting | ☒ charset |
| | CVE-2007-6422 | `mod_proxy_balancer` | null dereference | ∅ |
| | CVE-2007-6421 | `mod_proxy_balancer` | cross-site scripting | ☒ memory |
| | CVE-2007-6388 | `mod_status` | cross-site scripting | ☒ memory |
| | CVE-2007-5000 | `mod_imagemap` | cross-site scripting | ☒ charset |
| 2.2.6 | CVE-2007-3847 | `mod_proxy` | buffer overread | ✓ |
| | CVE-2006-5752 | `mod_status` | cross-site scripting | ☒ charset |
| | CVE-2007-3304 | MPM | SIGUSR1 killer | ☒ design |
| | CVE-2007-1862 | `mod_cache` | information leak | ✓ |
| | CVE-2007-1863 | `mod_cache` | null dereference | ∅ |
| 2.2.3 | CVE-2006-3747 | `mod_rewrite` | off-by-one | ✓ |
| 2.2.2 | CVE-2005-3357 | `mod_ssl` | null dereference | ∅ |
| | CVE-2005-3352 | `mod_imagemap` | cross-site scripting | ✓ |

∅ = auditing not required ✓ = RAD works ☒ = false alarms

**Figure 3: Vulnerabilities in Apache 2.2.x releases [1].**

changes every output, even when there is no compromise. RAD would report false alarms.

*Design flaw* (2). Two vulnerabilities involve design flaws. A malicious local user may override some permissions enforced in `httpd.conf`, or kill an arbitrary process by spoofing the scoreboard. The patches change either the semantics or the architecture of the Apache program. Whole-process auditing seems unavoidable since the changes are global, instead of local to a single function. For Apache, whole-process auditing can lead to false alarms due to non-determinism.

In summary, RAD may report false alarms for 9 of the 36 vulnerabilities in Apache. Specifically, 3 XSS cases require a more elaborate diff, 4 charset-XSS and CSRF cases need further incorporation with browsers, and the 2 design cases would require whole-process auditing. Section 5 speculates how we might handle these cases.

# 4. INITIAL RESULTS

This section presents the results of applying RAD to retroactively auditing two vulnerabilities: CVE-2009-0023 in Apache 2.2.10 and CVE-2005-3352 in Apache 2.2.0. For these two vulnerabilities RAD does not report false alarms for normal workloads, and is effective in detecting all exploits of the vulnerabilities. All the experiments are conducted on 64-bit Ubuntu 10.10 with Linux kernel 2.6.35.

## 4.1 Case study I: Heap underwrites

The first case study of retroactive auditing is CVE-2009-0023, the running example shown in Figure 2. The vulnerable function `apr_strmatch_precompile` is invoked by both the server core and several modules, such as `mod_substitute`, which uses it to perform string substitutions on HTTP response bodies. This vulnerability allows a malicious local user to mount an attack by creating an `.htaccess` configuration file that contains a bad string.

To evaluate whether RAD will generate false alarms we create two different `.htaccess` files and enable `.htaccess` in `httpd.conf`. The first file is created in a directory named `good`, which contains the following line:

```
Substitute s/work/sink/n
```

```
--- modules/mappers/mod_imagemap.c ...
+++ modules/mappers/mod_imagemap.c ...
@@ -342,6 +342,6 @@
 if (!strcasecmp(value, "referer")) {
   referer = apr_table_get(r->headers_in, "Referer");
   if (referer && *referer) {
-    return apr_pstrdup(r->pool, referer);
+    return apr_escape_html(r->pool, referer);
   }
```

**Figure 5: Part of the patch for CVE-2005-3352, cross-site scripting in mod_imagemap.**

A web page in this directory that originally displays "it works" will be rewritten to "it sinks", where the string "work" is used as input to `apr_strmatch_precompile`.

To simulate an attack we create another `.htaccess` file in a directory named `bad`; this file differs from the previous one in changing the character "o" in "work" to `0xf0`. This string tricks the vulnerable function into writing integer 2 into `shift[-16]` rather than `shift[240]`.

To generate a workload, we send 100 HTTP requests for files in the `good` directory. RAD records 110 invocations of `apr_strmatch_precompile`, among which are 100 calls from `mod_substitute`, and the remaining 10 are invoked from the server core and other modules. RAD audits each invocation and reports no warnings.

We also request one file in the `bad` directory. This time RAD reports one alarm and outputs the following diff of memory addresses and corresponding hex values:

```
            address  original  patched
0x00000000007ac188         02       --
0x00000000007ac189         00       --
0x00000000007ac18a         00       --
0x00000000007ac18b         00       --
0x00000000007ac18c         00       --
0x00000000007ac18d         00       --
0x00000000007ac18e         00       --
0x00000000007ac18f         00       --
====================================
0x00000000007ac988         04       02
```

The diff indicates that the vulnerable code wrote the value 2 to address 0x7ac188 (`shift[-16]`), which is the heap underwrite. The patched code did not write to this address. It also shows that the vulnerable code left 4 at 0x7ac988 (`shift[240]`), while the patched code wrote 2 there.

## 4.2 Case study II: Cross-site scripting

Figure 5 shows part of the patch for CVE-2005-3352, a vulnerability in `mod_imagemap`, a module for processing imagemap files. The vulnerable code copies the value of the "Referer" field to the HTTP response without sanitizing it, which allows attackers to launch cross-site scripting attacks. The patch escapes the value. Note that unlike the previous cross-site scripting case in Figure 4, in this case both the vulnerable and the patched code return a duplicated string, so the memory layout is unchanged by the patch.

To test RAD, we create an imagemap file served by the vulnerable Apache server, and set up a separate web site, hosting a normal web page `good.html`, and `bad.html`, which will exploit the vulnerability to steal a user's cookie. As a user, we visit both web pages using the IE browser from a Windows XP SP3 machine.

In the first experiment, a user visits `good.html` 10 times. RAD records 80 invocations to the vulnerable function in `mod_imagemap`, and audits each invocation by comparing the memory writes by the vulnerable and the patched code. It reports no warnings.

In the next experiment, the user visits `bad.html`. RAD audits the invocations to the vulnerable function, and reports one alarm. The important excerpt from the memory diff between the vulnerable and patched runs is as follows:

```
             address  original  patched
0x0000000001da2b11         >        &
0x0000000001da2b12         <        g
0x0000000001da2b13         s        t
0x0000000001da2b14         c        ;
0x0000000001da2b15         r        &
0x0000000001da2b16         i        l
0x0000000001da2b17         p        t
0x0000000001da2b18         t        ;
```

This excerpt shows part of the injected `<script>` tag that was passed through by the original code, and the corresponding escaped characters from the patched code.

# 5. FUTURE CHALLENGES

If an attacker exploits a vulnerability, RAD will identify it. The main challenge for RAD is avoiding false alarms. The preliminary experimental results for fine-grained auditing of two vulnerabilities are promising: RAD does not report any false alarms and identifies compromises correctly. The feasibility study further suggests that RAD should do well on many vulnerabilities: 15 out of 24 should not cause any false alarms.

To reduce false alarms for the remaining ones, we plan to refine the diff approach by comparing data structure topology, rather than the raw diffs of memory writes. We can consider two memory images to be equivalent if the corresponding graphs formed by memory blocks and pointers among them are equivalent, i.e., isomorphic, even if the addresses of memory blocks differ. This plan should eliminate false alarms caused by the XSS patches, as long as the patches do not introduce or remove any pointer aliasing (which would make the memory graphs non-isomorphic).

Another source of false alarms arises from client-side exploits, such as the charset-XSS and CSRF vulnerabilities. To determine whether they actually caused attacks at client side, we would like to integrate the browser behavior into the auditing system. For example, RAD could see whether the browser generates the same DOM tree for the HTTP responses from the web server with and without the patch.

In addition to memory differences, RAD should also record system calls issued during re-execution to determine if there are differences in the effects on the execution environment. To avoid conflicting changes to the system state, we expect that RAD would allow system calls from one of the forked processes (e.g., the unpatched parent) to proceed, and intercept system calls from the other fork (e.g., the patched child). If the child's system calls or arguments differ from those in the parent process, an alarm is raised and the child is terminated. If the child's system calls and arguments are the same, the return values from the parent's corresponding system calls are supplied to the child.

Finally, we would like to extend RAD to help the administrator determine exploit severity and analyze actual damages (e.g., determine which files an adversary has modified). We plan to leverage Retro [7] to address both issues.

# 6. RELATED WORK

Existing intrusion detection tools include COPS, Snort, and Tripwire [6], among many others [9]. These tools log system activities and report anomalies according to a set of predefined security rules. RAD takes a different approach: it uses patches to detect exploits of software vulnerabilities.

IntroVirt [5] asks developers to write predicates in addition to patches for every vulnerability. IntroVirt then detects past intrusions

by checking these predicates while replaying past executions of the system at the virtual machine level. For example, the predicate for the vulnerability in Figure 2 would be as follows:

$$\begin{aligned} &\text{For any character } c \text{ in } \texttt{s}, \\ &\qquad 0 \le c \le 127 \quad \text{if } \texttt{case\_sensitive} \text{ is true,} \\ &0 \le \texttt{tolower}(c) \le 127 \quad \text{otherwise.} \end{aligned}$$

RAD's new auditing scheme leverages patches to remove the burden of writing predicates.

Retro [7] uses re-execution to repair the system. It asks the administrator to mark the initial intrusion point, and re-executes legitimate actions to construct a new system state as if the intrusion never happened. RAD can use a system like Retro to roll back to an earlier state and replay inputs. It extends Retro by retroactively identifying compromises based on security patches. If RAD finds an attack, an administrator can then use Retro to recover.

Delta execution [11] is a mechanism to validate untrusted code changes by comparing system behaviors running different versions of the program. In contrast, RAD trusts security patches, and uses them to detect intrusions.

BinHunt [3] displays the differences between the original and the patched versions of a program by constructing their control flow graphs from instructions and comparing the two graphs. RAD further tells whether the two versions would behave the same given a particular input, and uses that information to infer whether a security vulnerability may have been exploited.

# 7. CONCLUSION

RAD allows an administrator to determine whether an attacker exploited a vulnerability. RAD audits the past execution of a system based on security patches released by vendors. It retroactively applies the patch and executes the patched code with the original input, and compares the resulting output with the original output. If the outputs are different, RAD concludes that the vulnerability may have been exploited. A feasibility study suggests that this approach is applicable to a complex program like the Apache web server.

## Acknowledgments

We thank the anonymous reviewers for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143. The opinions in this paper don't necessarily represent DARPA or official US policy.

## References

[1] Apache httpd 2.2 vulnerabilities. http://httpd.apache.org/security/vulnerabilities_22.html.

[2] B. Chelf and A. Chou. Controlling software complexity. http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf, 2008.

[3] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, Birmingham, UK, October 2008.

[4] E. R. Harold. Tip: Configure SAX parsers for secure processing. http://www.ibm.com/developerworks/xml/library/x-tipcfsx.html, 2005.

[5] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[6] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1994.

[7] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, October 2010.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[9] D. Swan. comp.os.linux.security FAQ. http://www.linuxsecurity.com/docs/colsfaq.html, 2002.

[10] L. Torvalds. Re: [RANT] Linux-IrDA status. http://lkml.org/lkml/2000/11/8/1, 2000.

[11] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.

# AutoLog: Facing Log Redundancy and Insufficiency

Cheng Zhang[1], Zhenyu Guo[2], Ming Wu[2], Longwen Lu[1], Yu Fan[1]
Jianjun Zhao[1], Zheng Zhang[2]
[1]Shanghai Jiao Tong University, [2]Microsoft Research Asia
{cheng.zhang.stap, nowen, yuf, zhao-jj}@sjtu.edu.cn
{Zhenyu.Guo, miw, Zheng.Zhang}@microsoft.com

## ABSTRACT

Logs are valuable for failure diagnosis and software debugging in practice. However, due to the ad-hoc style of inserting logging statements, the quality of logs can hardly be guaranteed. In case of a system failure, the log file may contain a large number of irrelevant logs, while crucial clues to the root cause may still be missing.

In this paper, we present an automated approach to log improvement based on the combination of information from program source code and textual logs. It selects the most relevant ones from an ocean of logs to help developers focus and reason along the causality chain, and generates additional informative logs to help developers discover the root causes of failures. We have conducted a preliminary case study using an implementation prototype to demonstrate the usefulness of our approach.

## 1. INTRODUCTION

Making complex software is hard, and getting them correct is often harder. Despite many progresses made by the research community, the adoption of advanced formal procedures and tools has been slow. Logging by *printf* is the predominant debugging practise, and will likely remain so for many years to come. In many ways, logging statements (i.e., the statements to print logs) can be collectively regarded as a "program" over the program under debugging. They represent the developers' effort to observe the inner working of the program, revealing expected or erroneous behavior. They are convenient to add, and do not create side effects, other than some (hopefully) minor runtime overhead.

This freedom comes with a cost. The practise of logging is ad-hoc. Logging statements accumulate over time by different developers, and the *quality* of the logs has never been a focus of either the research or the development community. Once hitting a bug, we are overwhelmed by the voluminous logs. Filtering by "grepping" reduces the overload to certain degree, but using the logs to reason about the causality is still difficult, since the relationships between the logs are obscure. Yet, many critical logs that could have led to quick discovery of the bug can still be missing.

Rooting out the bugs is ultimately the responsibility of the developer. Acknowledging the fact that logging is the critical debugging exercise, and that debugging is typically interactive, AutoLog attempts to provide a set of practical techniques to improve the utility of logs. More specifically, we make the following contributions:

1. *log slicing* combines program analysis techniques and information provided by the logs, and presents only the relevant (though conservative) logs, framed over causality chain.
2. if and when the developer requires further probing, *log refinement* inserts new logging statements, prioritized by their degree of uncertainty reduction.

This paper describes the prototype architecture of AutoLog. Our early experience of using it against a real-world complex software (Apache Hadoop Common) has shown the promise of the direction.

The rest of this paper is organized as follows. Section 2 gives an overview of our approach. Section 3 describes the technical details. Section 4 briefs the prototype implementation and shows the result of a case study. We discuss related work and our future plan in Section 5.

## 2. OVERVIEW

**Target scenario.** We imagine that AutoLog will be used in an interactive debugging session. Our target scenario is pre-release in-house development, where the source code is available. When the session starts, there is at least one log available, namely the one corresponds to the failure site. That starting log, which can be an error message, an assertion failure, or an exception, may or may not contain other rich information. For our prototype we have assumed that the stack trace and the variable that triggered the failure are available when the failure is reproduced for the first time. This is a constraint that we might remove in the future.

From that starting log, typically developers will explore backward, searching for an answer to explain why the bug has occurred; they might also explore along the execution flow forward to consolidate their understanding of the program behavior. When they cannot clearly pinpoint the root cause yet but have formed some clues, they may choose to insert additional logging statements in subsequent runs. Once they have identified the bug and installed the corresponding fix, the program is run again, and this cycle might repeat.

**Debugging using AutoLog.** Figure 1 shows the AutoLog architecture as well as the flow; our current prototype only

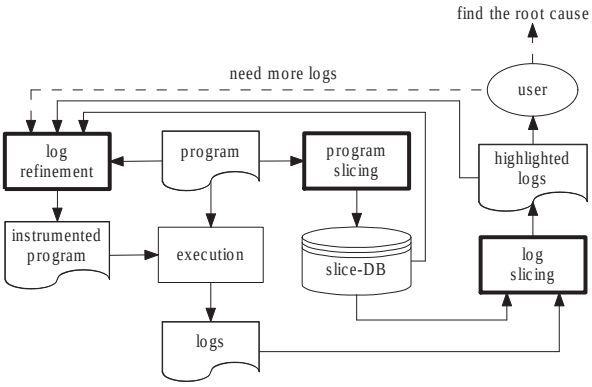


**Figure 1: The Architecture of AutoLog.**

contains part of the system. The components of AutoLog are called into play in different stages of the debugging session.

In the offline part of the process, AutoLog takes the source code and generates a conservative set of slices and stores them into a database (i.e., the slice-DB in the figure). A slice is the set of statements that may affect the value of a specific variable at a specific program location (the pair of variable and location is called *slicing criterion*). Slice-DB may contain only a subset of all possible program locations, as computing slice is expensive. Selecting good candidates for the slice-DB is itself an interesting topic. For now, we start with the logging statements, as they are inductive: the fact that the developer has added an entry there means that she wants to know more about the "what" at that point, and we might as well be prepare to answer the "why".

When the debugging session starts, AutoLog takes the starting log, and explores backwards from the closest entry in the slice-DB. During this step, it also scans through the logs, taking the runtime information that they might contain, and prunes the conservative slice(s) with an approach that we call *log slicing* (see Section 3). The end of this scan process will produce the set of logs that are aligned along the causality chain. Only these logs are presented to the developer. It is critical to optimize the performance of log slicing, as it directly affects the usability of the tool.

Typically, when the developer studied the logs and the relevant part of the code, she would have formed at least some clues as why the bug has appeared. If the exact root cause remains elusive, it means that more information is needed. New logging statements can be manually added, as is done today. AutoLog goes further, it computes and instruments new logging statements based on a set of heuristics to reduce ambiguity aggressively. This part of the process is called *log refinement*, and will be detailed in Section 3.

We believe that the above interactive debugging process is typical, in which AutoLog embeds naturally. This approach has a number of challenges, however. For instance, we assume that the bug is deterministic and can be triggered again. There is also the weakness associated with the slicing technique itself, such as the difficulty to handle aliases and shared variables across threads. We will delay the discussion on the challenges and limitations until Section 5.

# 3. APPROACH

The goal of AutoLog is to aid, not to replace, a programmer in her pursuit to discover and fix the bug. Since debugging involves reasoning about causality, AutoLog relies on slicing [5] as the underlying technology. The original program slicing is based on static (data and control) dependencies, thus the slice contains all the statements that **may** affect the slicing criterion in any possible execution. In contrast, dynamic slicing focuses on dependencies that occur in a specific execution. Therefore, the dynamic slice contains the statements that have **actually** affected the slicing criterion in one execution.

In the example shown in Figure 2, using statement 13 and variable `a` as slicing criterion, we get a static slice which consists of all the statements except statement 10. Statement 10 calls the logger which reads variable `a`; it could be included in the slice by data dependency if it wrote variable `a`. In an execution, if `p` is 1 and method `read()` returns 2 and 0 at statements 1 and 5, respectively, then the assertion at line 13 fails. In this case, the dynamic slice contains statements 2, 3, 4, 5, 6, 8, and 13.

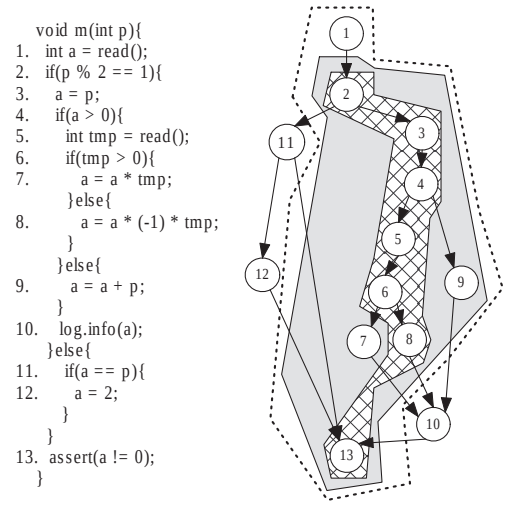


**Figure 2: A program and its control-flow graph. Nodes surrounded by dashed border represent the static slice pruned by control flow relations. Nodes with pattern/shaded background represent the dynamic/hybrid slice. Note that the data dependency graph, which is necessary for computing the slices, is omitted for brevity.**

Both static and dynamic slices rooted from the failure site (line 13) contain the culprit (line 5). However, the dynamic slice is far more precise. The tradeoff is that, while static slice can be computed offline, dynamic slice requires heavy instrumentation to acquire the entire execution trace, in addition to computation of the slice.

The core idea of AutoLog is to refine the scope of static slices using the dynamic information already made available from the printed logs. The advantage is that the expensive operations, such as computing dependency graphs and static slices, are strictly offline from a debugging perspective. Furthermore, when new logging statements are inserted, static slices do not change and can be reused. The online part of the refinement is hopefully very light weight, therefore maximizing the usability of the tool. To achieve this goal, log slicing in AutoLog is *hybrid*, in the sense that both static and dynamic information are used.

Given a log file, AutoLog first parses the textual logs and maps them to the corresponding logging statements. Cur-

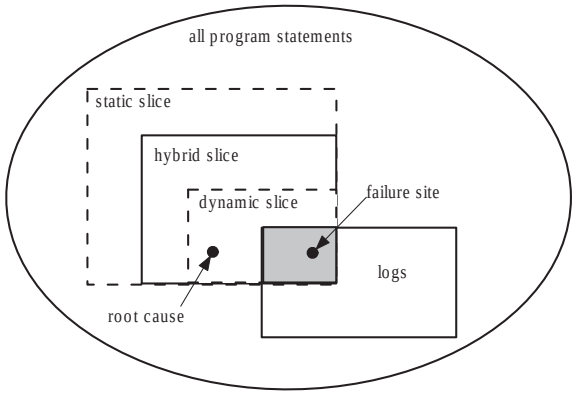


**Figure 3: The relationship between different kinds of slices and logs.**

rently we assume that the logs are produced with general logging frameworks, such as log4j, so that the exact locations of logging statements can be included in logs and easily parsed from logs. With this assumption, the mapping between logs and logging statements is straightforward. However, as discussed in prior work [6, 7], a more general log parsing technique is often necessary, since many systems actually have their own logging mechanisms. We are developing a log parser targeting object-oriented programs. It is an extension to the log parser proposed in [6] and will address the complexity of string manipulation based on [2].

The goal of AutoLog's log slicing is to classify statements in the corresponding static slice into groups of **must**, **may**, and **must not** and then refine the slice. This is done in two steps: 1) pruning and 2) re-slicing. The pruning step filters out from the static slice the statements that must not have been executed. This step is trivial to compute, an essentially reachability analysis starting from the executed logging statements. In Figure 2, the log printed by node 10 will prune away nodes 11 and 12. We can also conclude that nodes 1, 2, 3, 4, and 13 must have been executed, because nodes 1, 2, 3, and 4 dominate node 10 and node 13 post-dominates node 10. These results are useful already; an experienced developer might have already concluded that statement 5 is where the problem is.

In the re-slicing step, the hybrid slice is computed using the pruned static slice as well as the control-flow graph. In Figure 2, because nodes 11 and 12 are excluded, the assignment at node 12 can never affect the slicing criterion. In addition, as one branch of node 2 has been pruned, the assignment at node 1 can no longer affect the slicing criterion (the definition is "killed" by the assignment at node 3). As a result, the hybrid slice is {2, 3, 4, 5, 6, 7, 8, 9, 13}. Compared with the original static slice, the hybrid slice contains three statements fewer (namely, statements 1, 11, and 12). This reduction is achieved with purely the position of the executed logging statement. Moreover, the observed value in the logs can be picked up by a constraint solver, which potentially refines the slice further, as is done in SherLog [7]. In the example, using the value exposed at node 10 (i.e., `a`, which is 0), node 9 can be excluded too. Investigating the utility of a constraint solver is one of our future works.

Figure 3 illustrates the relationship between logs and the different kinds of slices (namely static, dynamic, and hybrid slices) with respect to a given slicing criterion. Static and dynamic slices are represented by squares with dashed borders, since AutoLog does not directly present the entire static slice, nor does it attempt to capture the dynamic slice. Obviously, hybrid slice is a subset of static slice and a superset of dynamic slice. The common property of these slices is that they all contain statements that may affect the slicing criterion. This property does not hold for logs, since there may be a lot of logs that are irrelevant to the failure. The shaded rectangle represents the intersection of dynamic slice and logs[1]. Only the logs in this rectangle will be presented to the developer, aligned with statements in the static slice. In other words, AutoLog intrinsically solves the problem of information overloading caused by irrelevant logs.

However, the sparsity of logs in practice might lead to information "underloading" instead, that is, there are missing logs that could have helped to pinpoint the bug. Conceptually, this happens when the root cause is not "covered" by the existing logs, as shown in Figure 3. A developer might add new logging statements, and then rerun the test. In log refinement, AutoLog automates this step by proposing and adding new logging statements. The net effect is to enlarge the set of logs (i.e., the lower right rectangle in Figure 3), while shrinking the hybrid slice towards the dynamic slice to cover the failure's root cause ($RC$). Since the dynamic slice consists of all the statements that are actually relevant to the failure, it must contain $RC$. In addition, as the hybrid slice ($HS$) is a superset of the dynamic slice, it certainly contains $RC$, too. When $RC$ is not contained in the highlighted logs ($HL$), it must belong to the set $R = HS - HL$.

AutoLog iteratively selects some locations from $R$ to insert new logging statements and re-executes the program to generate new logs. We follow a group of heuristic rules, with the goal to approach the root cause quickly with fewer logs:

**Rule of loop avoidance.** If a new logging statement is inserted in a loop, then it is likely to be executed for multiple times, which generates bloated log files. This rule avoids inserting logging statements in loops. More specifically, AutoLog calculates the depth of nested loops where candidate statements reside. Then the statements are sorted by the depth in an ascending order.

**Rule of control-flow uncertainty reduction.** During debugging it is often helpful to know whether certain statements have been executed in the failing run. In order to reduce the uncertainty of control-flow, AutoLog inserts new logging statements at the statements (in $R$) that **may** have been executed. Moreover, AutoLog prefers to choose the statements with higher power of uncertainty reduction. As previously described, the execution of one statement can be used to determine a group of statements which must have been executed (denoted as $M$) as well as a group of statements which must not have been executed (denoted as $MN$). When applying this rule, AutoLog first calculates the number $|M \bigcup MN|$ for each candidate statement to represent its power of uncertainty reduction and then sorts the candidates by the number in a descending order. In our example, the number $|M \bigcup MN|$ for nodes 5, 6, 7, 8, 9 is 2, 2, 4, 4, 4, respectively[2]. Therefore, AutoLog ranks nodes 7, 8, and 9 higher than nodes 5 and 6.

[1]More exactly the shaded rectangle represents the logging statements that are control-equivalent to some statements in the dynamic slice.

[2]For instance, for node 7, $M$ is {5, 6} and $MN$ is {8, 9}, thus its $|M \bigcup MN|$ is 4.

```
1: function RefineLogInMethod(method)
2:     for all stmt_i ∈ method.stmts do
3:         calculate its depth of loops, d_i
4:         calculate its number n_i = |M_i ⋃ MN_i|
5:         calculate its number of interesting variables, m_i
6:     end for
7:
8:     NumStmts ← method.stmts.size
9:     sort method.stmts by d_k, 1 ≤ k ≤ NumStmts, in an
       ascending order
10:    sort the tying statements in method.stmts by n_k, 1 ≤
       k ≤ NumStmts, in a descending order
11:    sort the tying statements in method.stmts by m_k, 1 ≤
       k ≤ NumStmts, in a descending order
12:    choose the top one statement, topStmt
13:    compute the interesting variables at topStmt, vars
14:
15:    create a new logging statement logStmt
16:    logStmt.location ← topStmt
17:    logStmt.variables ← vars
18:    insert logStmt into method.code
19: end function
```

**Figure 4: Algorithm of log refinement for one method**

**Rule of value uncertainty reduction.** If a logging statement resides at a location at which many relevant variables are accessible, then it can reveal valuable information of runtime program state by printing the values of these variables. When applying this rule, AutoLog first calculates the set of visible interesting variables for each candidate statement. A variable is said to be interesting if it is in the data flow set before the statement in the data flow analysis during program slicing, that is, its value may be relevant to the slicing criterion. Then AutoLog sorts the candidates by the number of visible interesting variables in a descending order. In the example in Figure 2, since the interesting variable `tmp` is accessible at nodes 7 and 8 (but not at node 9), AutoLog prefers to choose either of nodes 7 and 8 as the candidate location to insert a new logging statement.

AutoLog applies the rules in the order they are described[3] and selects the location to insert the new logging statement from the top statement(s) for each method. If there are multiple top statements, AutoLog randomly chooses one of them. Therefore, in each iteration AutoLog inserts at most **one** new logging statement in each method to control the total amount of logs that will be generated. We also take care to print variables that have already been initialized at the new logging points, based on data flow information generated during program slicing. Figure 4 shows the algorithm of log refinement for a method.

**Trade-offs.** In theory, slices are computed using both control and data dependencies in order to conservatively include all the statements which may cause the failure. In practice, as discussed in [4], many bugs can be discovered by exploring data dependency. Moreover, control dependency can be converted to data dependency of the corresponding branching statements in structured programs. Therefore, AutoLog makes the trade-off between precision and efficiency, in that it does not use control dependency during slicing. As a result, the slices may fail to include clues to bugs, but the size of slices can be reduced. In case the developer finds that a control dependency is too important to ignore, she can manually specify the corresponding branching statement and variable as slicing criterion, and re-launch AutoLog.

The hybrid slicing algorithm of AutoLog is inter-procedural, that is, it will track data dependencies across procedure boundaries by following calling relations. When performing slicing in a method `m`, AutoLog will explore forward into the relevant methods called by `m` and backward into the methods that call `m`. The inter-procedural slicing is necessary for AutoLog to capture the root cause which does not reside in the same method with the error-reporting logging statement. In order to allow the hybrid slicing to handle large-scale systems, we impose two restrictions on the slicing procedure:

- In the forward exploration, the slicing procedure does not explore further along a call chain if a cycle is encountered (i.e., there are recursive calls).
- We augment each logging statement with an extra statement to record the stack trace. In this way, we can obtain the exact runtime stack trace of the starting log. Therefore, when performing backward exploration, the slicing procedure only has to explore the methods along the recorded stack trace, instead of all the call chains leading to the current method. Note that we just have to record the stack trace during the first execution in which the failure is reproduced. Once the failure has been observed, we can turn off the stack trace recording to reduce overhead in subsequent re-executions.

# 4. PRELIMINARY CASE STUDY

We have implemented a prototype of AutoLog on top of the Soot framework[4] which supports various program analyses on Java programs. We have not implemented the offline part that prepares the slice-DB. Rather, when performing log slicing, the slice is computed from scratch.

We now describe a proof-of-concept case study on Apache Hadoop Common, the core sub-project of the Apache Hadoop distributed computing framework[5].

In the case study, we have used the prototype to debug a reported bug[6] for Hadoop Common version 0.19.0. The bug manifests itself as a reproducible test case failure caused by an `IOException`. The bug is in the method `listStatus` of class `SequenceFileInputFormat`, a subclass of class `FileInputFormat`. The correct behavior of the method is to return the list of files contained in the variable `file`, if `file` is a directory. However, if one of the files is indeed a directory, rather than returning the status of that directory, the buggy implementation returns a new file object with attribute `isdir` as true.

The exception is thrown from method `getSplits` in class `FileInputFormat`. Before splitting files, the method checks whether the files, retrieved via a call to method `listStatus`, are actually files rather than directories. The tricky part is that class `FileInputFormat` has its own implementation of method `listStatus`, which has been overridden in class `SequenceFileInputFormat`. Thus the method call to method `listStatus` in method `getSplits` have two possible targets, and only when the underlying object is an instance of class `SequenceFileInputFormat`[7], the `IOException` will occur.

[3]Specifically, the application of a rule may only change the order of the candidate statements that have been ranked the same by the previous rule.

[4]http://www.sable.mcgill.ca/soot/

[5]http://hadoop.apache.org/

[6]https://issues.apache.org/jira/browse/HADOOP-3946

[7]Besides `SequenceFileInputFormat`, there are several subclasses of `FileInputFormat`.

In the original log file generated by the failed test run, there were 86 lines of logs. AutoLog highlighted 17 logs, containing the most relevant logs printed in method `listStatus` of class `FileInputFormat`. The rest of the logs were mostly about the progress of map/reduce tasks, which were irrelevant to the exception. However, the 17 highlighted logs might be insufficient for finding the bug, and even a bit misleading. Because the last line of log was printed in method `listStatus` of class `FileInputFormat`, developers might conclude that the erroneous files were returned from this method. Since the implementation of this method is complex, developers might waste a lot of time and fail to find the real bug in method `listStatus` of class `SequenceFileInputFormat`. Therefore, we used the prototype to insert new logging statements and re-ran the test case. After two iterations of refinement, the logs successfully covered the root cause. The last two lines of logs were printed from method `listStatus` of class `SequenceFileInputFormat` and method `isDir` of class `FileStatus`, which made the bug quite obvious.

Although the prototype succeeded in revealing the bug, the final log file contained as many as 1778 lines of logs. It was mainly due to our unsophisticated rules of log refinement. For example, the current rule only tries to avoid inserting logging statements in explicit loop structures in the scope of a method. However, in the case study, a number of logging statements were inserted into methods that were called from within loops in their callers.

The case study was conducted on a Linux server, which has a 2.33GHz quad-core CPU and 16 GB main memory. The average runtime of each iteration (including log slicing and log refinement) is about 11 minutes. For Hadoop Common, which has more than 143K lines of code, the runtime is probably acceptable. Our existing prototype does not allow easy estimation of how much time can be moved to offline. However, we are optimistic that a good portion will be.

# 5. DISCUSSION

The exercise of debugging asks the programmer to act as a detective. Having arrived at the crime scene, she needs to reconstruct the sequences of events, reasoning about the "why" and "how" along the way. Building a time machine that allows replaying the entire execution path represents one type of tools. The spectrum includes tools such as R2 [3] that relies on instrumentation at the boundary of nondeterminism, to SherLog [7] that infers the paths with the combined knowledge of runtime logs and program structure (via constraint solving).

Replaying tools narrow down the search space to concrete execution instances, but do not solve the problem of constructing the path that connects failure site to root cause. In addition, knowing the entire execution history may not be necessary.

At the other extreme of the spectrum, there are bug detection tools that try to directly identify common mistakes (e.g., use-after-free [1]). These faulty patterns are shortcuts to directly flag not only the manifested bugs but also the potential ones. However, we argue that discovering causality interactively is the most common case, and needs better tools. A natural starting point is to leverage the technique of slicing [5].

AutoLog is somewhere in between. We target the scenario of interactive in-house development, in which quick turnaround time is critical. This is achieved by splitting offline static slice preparation from online log slicing and refinement. We share the same philosophy as LogEnhancer [8], in that the existing logs are treated as heuristics and starting points, and a tool is free to modify them, as well as to insert new ones.

Our future work has a few focuses:

1. For log slicing, simply pruning the static slices with log positions, performing hybrid slicing, and calling a constraint solver incrementally improves coverage, with increasingly higher cost and longer delay. The trade-offs there are complex, and we plan to quantify them with a set of real examples.

2. The heuristics used in the current log refinement algorithm is neither sophisticated nor well tested. Again, this requires in-depth study over real examples.

3. Once the programmer formed concrete clues and installed new fixes, the static slices need to reflect these changes. Improving the productivity requires us to recompute the slice-DB incrementally and efficiently.

4. The above process works well when non-determinism is absent. This isn't the case for many concurrent programs. We see the role of replay tools to remove the randomness so that AutoLog can do its work with as little as (and ideally no) uncertainty. How to leverage these techniques is still an open question.

5. An orthogonal but important question is to understand and improve the quality of the logs. If and when AutoLog is continuously applied throughout the development process, it is possible to handle and manage the logs better. As we mentioned earlier, the framework of AutoLog can already detect redundant logs.

# 6. REFERENCES


[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *OSDI'08*, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[4] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[5] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[7] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. ASPLOS '10, pages 143–154, New York, NY, USA, 2010. ACM.

[8] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. ASPLOS '11, pages 3–14, New York, NY, USA, 2011. ACM.

# Hardware-Supported Virtualization on ARM

Prashant Varanasi
prashant.varanasi@gmail.com

Gernot Heiser
gernot@nicta.com.au

NICTA, University of New South Wales and Open Kernel Labs
Sydney, Australia

## ABSTRACT

ARM is the dominant processor architecture for mobile devices and many other high-end embedded systems. Late last year ARM announced architectural support for virtualization, which will allow execution of unmodified guest operating system binaries. We have designed and implemented what we believe is the first hypervisor supporting pure virtualization using those hardware extensions and evaluated it on simulated hardware. We describe our approach and report our initial experience with the architecture.

## Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

## General Terms

Design

## Keywords

Hypervisors, virtual machines, architecture, hardware support, ARM

## 1. INTRODUCTION

Virtualization, formerly mostly at home in data centres and enterprise computing infrastructure, is now spreading to embedded systems, driven by cost and security concerns [Hei08, Kro09]. ARM, the dominant architecture of high-end processors for mobile devices, is not trap-and-emulate virtualizable. This means that virtualization of ARM processors requires binary translation or para-virtualization. Binary translation is generally too resource intensive for mobile devices, which is why all known commercial and research hypervisors for ARM use this approach. Para-virtualization has the drawback of high engineering cost for having to adapt each supported operating system (OS) to the hypervisor-specific platform interface.

In the server world, which is dominated by the (also not trap-and-emulate virtualizable) x86 architecture, the virtualization boom lead hardware extensions to support virtualization [NSL+06]. These allow running an unmodified, native OS binary in a virtual machine (VM) with minimal performance degradation, and greatly simplify the implementation of hypervisors and reduce run-time overheads.

The same is now happening with embedded processors: last year, ARM announced virtualization extensions for their architecture [ARM10], along similar lines as the manufacturers of x86 processors.


*APSys'11* Shanghai, China


In this paper we present the first hypervisor which uses these extensions to support pure virtualization on ARM, and is able to run multiple concurrent unmodified Linux guests. We report on our experience with using the new extensions. Unfortunately, only extremely limited performance evaluation is possible, as the hardware extensions are presently only available in a simulator which is not timing-accurate.

The rest of the paper is structured as follows. Section 2 outlines existing work. Section 3 presents an overview of the ARM architecture, the virtualization extensions, and a comparison to x86 approach. Section 4 outlines the design and Section 5 presents the implementation of our hypervisor. We show TCB size and indicative performance numbers for our hypervisor in Section 6. We discuss our experience with the extensions in Section 7, and draw our conclusions in Section 8.

## 2. RELATED WORK

Commercial virtualization solutions for ARM platforms are provided by Open Kernel Labs [OKL11], VMware [VMwa10] and Red Bend Software [RedB10], these all use para-virtualization. Green Hills Software's Integrity product [Gree10] uses the TrustZone features of the ARM architecture to run a native guest binary, but architecture limitations restrict this to a single guest.

A port of Xen to ARM was performed by Samsung [HSH+08], but performance is poor: a Linux guest runs at about half of native speed. In contrast, the OKL4 microvisor from OK Labs, which is the only commercial product for which performance data is available, exhibits overheads which are about an order of magnitude lower [HL10]. NOVA [SK10] is a hypervisor for x86 which, like the OKL4 microvisor and our current design, uses a microkernel architecture aimed at minimising the *trusted computing base* TCB of virtual machines (VMs).

Fisher-Ogden presented a thorough analysis of virtualization extensions for x86 from Intel and AMD [FO06]. Adams and Agesen [AA06] found that binary translation outperformed pure virtualization, but this evaluation was completed before the hardware extensions included MMU virtualization. A later evaluation [Bha09] found MMU virtualization significantly reduced overheads, especially when using large pages. The ARM virtualization extensions already include MMU virtualization.

## 3. ARM ARCHITECTURE

The ARM architecture has evolved over the decades. Here we focus on the latest version, v7, which is the one for which the virtualization extensions are specified.

### 3.1 Overview

ARM is a 32-bit RISC architecture, featuring 16 general-purpose (GP) registers (which includes the program counter). There is one unprivileged processor mode (user) and six privileged kernel modes. All kernel modes have the same level of privilege, they differ in the kind of exception which forces their entry, the exceptions allowed while executing in them, and the number of banked registers.

The architecture supports a feature called *TrustZone*, which provides an orthogonal processor mode, called *secure mode*. Hardware resources

(memory and on-chip devices) are configured to be accessible always or only in secure mode. A super-privileged mode, called *monitor mode*, is used to switch between secure and insecure mode. On power-up the processor enters secure kernel mode. TrustZone can be used to run an unmodified native OS binary next to other native code, similar to virtualization. This is achieved by running the guest in non-secure mode and all other code in secure mode. Unlike real virtualization, this only supports a single guest.

The standard "ARM" instruction set uses 32-bit instructions. Notable features are predication of all instructions, and a barrel shifter which supports complex indexing and address-register updates. A system co-processor contains the MMU, cache control and the performance-monitoring unit (PMU). Further (up to 15) co-processors can be used to implement optional functionality, such as the FPU. There are two further instruction sets: Thumb-2, which uses variable-width (16- and 32-bit) instructions, achieves higher code density at a performance close to that of the ARM instruction set. A third instruction set, Jazelle, is designed for efficient execution of Java bytecode.

There are two levels of on-chip cache: a virtually indexed, physically tagged split I/D-L1, and a unified physically-addressed L2. The TLB is hardware-loaded from a two-level page table (PT) with 32-bit entries. TLB entries are tagged with an 8-bit *address-space ID* (ASID). Supported page sizes are 4KiB, 64KiB and 1 MiB. I/O is memory-mapped.

The architecture defines a *generic interrupt controller* (GIC) which contains two parts. The per-CPU GIC *CPU interface* performs interrupt priority masking and preemption handling, while the global *distributor* receives interrupts and controls interrupt priorities, enabling and the CPU interface to which it is routed. Software acknowledges an interrupt to the interface.

A number of unprivileged instructions reveal privileged information, thus preventing ARM from being trap-and-emulate virtualizable. For example, the `cps` (change processor state) instruction is silently ignored in user mode. In addition, while traps on ARM are relatively cheap (kernel entry- and exit costs are about a dozen cycles on ARM, compared to hundreds of cycles on x86), pure virtualization, if it was possible, would still have a high overhead. The reason is that modern OSes tend to perform frequent manipulations of privileged state, such as processor status and page table.

## 3.2 Virtualization extensions

The virtualization extensions [ARM10] to the ARM architecture are superficially similar to those for x86, in that they provide a new processor mode and a number features to improve performance.

The extensions only apply to non-secure mode. They introduce a new processor mode, *hyp* mode, which is more privileged than the existing non-secure kernel modes. This leaves the existing kernel and user modes for unmodified guest OSes and applications, as shown in Figure 1.

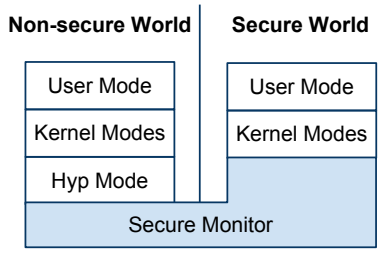


**Figure 1: ARM processor modes**

Hyp mode is entered from kernel mode via a new instruction (`hvc`), and optionally on a configurable set of exceptions from user or kernel mode. It has banked registers, as well as additional hyp-only registers for system configuration and information on the event which caused entry of hyp mode. There is a hyp-only *virtual machine identifier (VMID)* register. TLB entries are tagged with the VMID, which supports co-existence of mappings from multiple guests and thus eliminates the need to flush the TLB on a world switch.

The hypervisor's own PTs (for translating guest-physical as well as hypervisor-virtual addresses) use a new, wider format which supports 64-bit physical addresses (and has the potential to support 64-bit virtual addresses in the future). It increases the largest page size from 1 MiB to 2 MiB. The new format can optionally be used by the guest as well.

ARM supports a number of mechanisms for reducing the cost of pure virtualization, which we discuss now.

### 3.2.1 Configurable traps

Many exceptions can be configured to trap either into the hypervisor or the guest kernel. This drastically reduces the frequency of hypervisor entries, e.g. by configuring the syscall trap to be handled directly by the guest. Interrupt traps are configured globally, so either all interrupts invoke the hypervisor, or all are delivered directly to the guest of the presently active VM.

### 3.2.2 Emulation support

Load and store instructions are not inherently virtualization-sensitive, but become sensitive when operating on privileged data (eg. device registers). The hypervisor must decode such an instruction and emulate it. The overhead of emulation is not just the extra instructions executed (which include translating guest physical to physical addresses) but also the D-cache miss generated when loading the offending instruction (despite the fact that it has already been fetched into the instruction register and the I-cache). ARM's emulation support in most cases eliminates both the load and the software decode, by keeping the relevant information in hypervisor registers (source or target registers, whether it was a load or a store, the size of the data item to be transferred etc.)

### 3.2.3 Second-stage translation

Similar to extended PTs on x86, ARM supports two-stage address translation: guest-virtual to guest-physical (called *intermediate physical* by ARM) followed by guest-physical to physical. On a TLB miss in non-hyp mode, the hardware page-table walker traverses first the guest and then the hypervisor PT, and constructs a TLB entry representing the guest-virtual to physical translation. While this requires traversal of four levels of page tables, this can be reduced to three if the hypervisor uses 2 MiB superpages. Obviously, only a single translation stage is used when running in hyp mode.

### 3.2.4 Virtual interrupts

In order to avoid emulation of the interrupt controller (which would add significant complexity and require frequent traps into hyp mode), ARM introduced the concept of virtual interrupts. It is supported by a new hardware component, the *virtual CPU (VCPU) interface*. This can be mapped into the guest as the GIC CPU interface, and can be used by the guest to acknowledge and clear interrupts without trapping into the hypervisor. The hypervisor must still emulate the interrupt distributor; all guest accesses to it trap. This is not expected to cause performance issues, as the distributor is normally only accessed at boot time (or module load time) to register drivers for particular interrupts and route them to specific (virtual) CPUs.

If interrupts are configured to be handled by the hypervisor, the hypervisor can explicitly forward the interrupt to the current guest by raising the appropriate virtual interrupt on the guest's VCPU interface. A virtual interrupt can be linked to the physical interrupt, in which case the physical interrupt will be cleared without hypervisor intervention when the guest clears its virtual interrupt. The sequence of events is shown in Figure 2. The hardware drops the current interrupt priority during virtual interrupt processing, in order to allow further interrupts of the same priority, which may be destined for other VMs.

## 3.3 Comparison to x86

Unlike x86, where VT-x root mode is orthogonal to the existing protection rings, ARM's hyp mode is strictly more privileged than the ex-

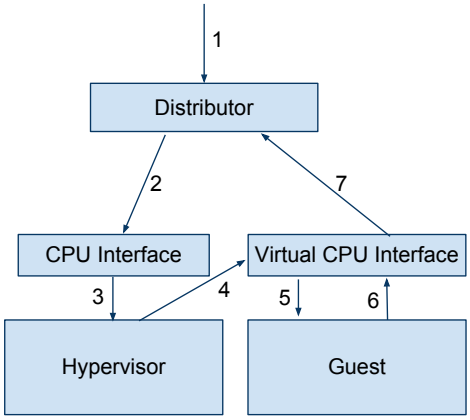


**Figure 2: Virtual interrupt processing (actions 4 and 6 are initiated by software).**

isting kernel modes. ARM requires the hypervisor to save guest register state, while on x86 this is done automatically by hardware. Second-stage translation and TLB tagging with a VMID are similar. While both architectures allow the hypervisor to inject interrupts into a running guest, ARM allows the guest to acknowledge, clear and mask interrupts without trapping; on x86 these trap into the hypervisor.

ARM's emulation support mostly frees the hypervisor from having to load and decode sensitive instructions. In contrast, the CISC nature of the x86 ISA does not support such a simple and elegant solution, and requires a complete ISA emulator in the hypervisor, adding significant run-time overhead as well as tens of thousands of lines of code (LOC) to the hypervisor [SK10].

ARM will support I/O virtualization via a *system MMU* (SMMU) [ARM11] similar to the IOMMU on x86, but this is a platform feature, and ARM has to date only released their extensions to the core ISA.

# 4. HYPERVISOR DESIGN

## 4.1 General

Our hypervisor supports a fixed, statically-configured number of VMs. Dynamic creation and destruction of VMs is a complication which reveals nothing of interest about the architecture, and therefore has no benefit for the prototype. Furthermore, virtualization use cases in the embedded-systems world typically do not require dynamic VMs, which is why they are not even supported by the commercial OKL4 microvisor. Hence even a productised successor of our prototype is likely to have this restriction.

We also decided against the use of virtual memory inside the hypervisor, as it would provide no benefit (especially when not supporting dynamic VMs). In fact, not using virtual memory has performance benefits, by avoiding PT walks inside the hypervisor and preventing the hypervisor from competing for TLB entries. Of course, virtual memory is available for use by guest OSes without limitation, and running the hypervisor in physical memory does not eliminate the need to set up PTs for the translation of guest-physical addresses.

As the OKL4 microvisor [HL10], and unlike other microkernel-like hypervisors (e.g. Nova [SK10]), we do not split the hypervisor into a privileged and a user-mode part. Nova, like other x86 hypervisors, require large amounts of code for instruction emulation, device emulation and BIOS emulation; separating these into user-level modules enhances robustness. In contrast, instruction emulation on ARM is almost trivial (thanks to the RISC architecture and architectural support), and there is no BIOS to emulate. Furthermore, thanks to virtual interrupts, the bulk of functionality required for device emulation is emulation of the distributor component of the GIC, which needs to be done in hyp mode anyway. Hence there is little functionality which could sensibly be moved to user-mode, and the hypervisor can be kept small nevertheless.

These design decisions helped to keep the hypervisor simple (without limiting its use in embedded systems). Beyond that we made one design decision which would be an unacceptable shortcut for a production system: not to support multiple cores. This is acceptable for the prototype as it avoids unnecessary complexity for the purpose of our evaluation, as multicore support would be mostly orthogonal to the use of the virtualization architecture. The structure of our kernel should make adding multicore support relatively straightforward.

## 4.2 Inter-VM communication

In addition to standard hypervisor functionality, our prototype provides a high-performance *inter-VM communication* (IVC) mechanism and the ability to set up shared buffers. This is essential to support shared device drivers encapsulated in their own VM, as well as the tight integration of subsystems typical of embedded systems [Hei09]. Such a mechanism is also provided by the OKL4 microvisor [HL10].

As in the microvisor, and unlike most other microkernels, IVC is fully asynchronous: A short (three-word) message is deposited in virtual registers in the receiver VM, and a virtual interrupt is delivered to the receiver. Further sends to the same receiver fail until the receiver acknowledges the interrupt. For the prototype we do not implement access control for IVCs, other than requiring the sender to know the receiver's VMID. Broadcast send to all other VMs is also supported.

A shared buffer needs to be set up between VMs at system-configuration time. A VM may have read-only or read-write access to a shared buffer. Within the bounds of this limited form of protection, VMs are fully responsible to manage access to a shared buffer. They can use virtual interrupts for synchronisation.

Remember that this simple model of IVC is designed for evaluation purposes. For a production system, more sophisticated access-control will be desirable, most likely an adaptation of the capability-based protection system of the OKL4 microvisor [HL10]. Furthermore, the message-passing semantics would need to be refined to avoid creating covert channels.

# 5. IMPLEMENTATION

The above design choices allowed us to based our implementation on an existing code base, the OK Labs "pico" product, which is an executive for MMU-less microcontrollers. No hardware supporting the virtualization extensions is available yet, so we used the ARM *Fast Models* simulator, which models the *RealView* emulation baseboard plus the virtualization extensions. The simulator is efficient yet functionally very accurate, but it is not timing-accurate. Also, it does not model complex external devices such as Ethernet, so we only support simple devices such as the UART consoles, the LCD controller, and the on-board timers. Here we highlight some implementation details.

## 5.1 World switch

A context switch performed by an OS needs to switch the GP registers plus the PT pointer and the address-space ID register, a total of 18 registers. The FPU state (16 or 32 double-word registers) is normally switched lazily. A world switch is more heavyweight, as it requires handling of additional state: the banked registers for all kernel modes (21), the system coprocessor registers, including the MMU state (22), all VCPU interface (interrupt controller) state (3 plus virtual interrupt registers, up to 96), all timer state (20), the second-stage PT pointer and the VMID register. In total this is up to 164 additional registers. Furthermore, co-processor registers and device registers are more expensive to access than GP registers.

World switches occur as a result of a timer tick (indicating the end of a time slice for a VM running on a shared core) and on an interrupt of priority higher than that of the presently running VM. The extensions introduce new timers which can be used by the hypervisor without interfering with the guests' use of timers. In our prototype we did not implement VM priorities and hence only perform world switches at the

expiry of a (33 ms) time slice.

## 5.2 Instruction emulation

For most instructions which trap into the hypervisor, the information provided in hypervisor registers is sufficient to emulate them. One exception is an ARM peculiarity known as *write-back*. It allows adding an immediate value to the address register in a load instruction. For example, the instruction

```
ldr   rd, [rs], #imm
```

is equivalent to

```
ldr   rd, [rs]
add   rs, #imm
```

except that it is atomic and executes as fast as a normal load.

Write-back is not supported by the emulation support. Also, the (previously fetched) instruction is not made available to the hypervisor, so it must be loaded explicitly from guest memory, decoded and emulated, and the guest instruction pointer must be incremented before returning to the guest. A very simple (50 LOC) emulator for write-back instructions was sufficient to support all Linux code we tried to run in a VM.

Note that loading the faulting instruction will cause at an L1 D-cache miss, as on fetch the instruction would have been placed in the I-cache.

## 5.3 Device pass-through

Pass-through allows a device to be exclusively used by a single guest, a situation common in embedded systems, where many devices are owned by individual subsystems. This does not incur any virtualization overhead and usually only requires mapping the correct memory regions, and ensuring that the device interrupt is routed to the guest. We used this for a range of devices, including the SMC flash chip, the network controller, and the audio controller.

Some cases require a bit more work, for example the LCD controller, which requires a physically-contiguous buffer and translating guest physical to physical addresses. In our implementation this is simple, as we map all guest physical memory (other than shared buffers) onto a contiguous memory region and trap the write to the buffer pointer register in the controller.

## 5.4 Shared devices

So far we have only added support for some very simple shared devices: a console (UART0), the distributor component of the GIC, the SP804 dual timer and the PL031 PrimeCell real-time clock.

The virtual console device runs in a VM in user mode on top of a real-time OS (we use the microvisor for this). It can be accessed from other VMs via an IVC protocol using a shared (single-page) buffer for transferring the data, and a virtual interrupt to indicate to the driver that data is available. Completion is indicated by a virtual interrupt back to the client.

We world-switch timer state, which means that guests see virtual time rather than real time. This is appropriate in many circumstances even in embedded systems (a trivial example are the BogoMIPS calculation loops in Linux).

We provide a second-resolution real-time clock by wrapping the hardware RTC to produce ticks every second, and sending virtual interrupts to all VMs that have enabled the RTC device. VMs with real-time requirements would be allocated a timer as pass-through device, but we did not do this in our prototype.

## 5.5 Limitations

The main limitations of our prototype are the lack of VM priorities (VM scheduling is round-robin) and no support for multiple cores. These limitations will be removed in a forthcoming production version of the hypervisor.

Other limitations do not affect anticipated embedded-systems use cases (at least in the near future). One is the lack of support for dynamic creation and destruction of VMs; in most embedded use cases the number of VMs are fixed at least between firmware upgrades. Similarly, there is little benefit expected from deduplicating pages, as embedded use cases tend to be heterogenous, running different guest OSes in different VMs (eg. Linux and an RTOS) [Hei08].

**Table 1: Estimated instruction latencies**

| Operation | Estimated cycles |
|---|---|
| L1 access | 2-3 |
| L2 access | 10-15 |
| Memory access | 50-150 |
| CP15 read | 10-20 |
| CP15 write | 100 |
| FPU access | 10-20 |
| Hyp entry | 50 |

# 6. EVALUATION

## 6.1 Hypervisor size

Our prototype comprises 5,730 LOC. We estimate that VM priorities would add at most 500 LOC, while multicore support would add about 1,000 LOC. This is offset by removing about 1,600 LOC of unneeded functionality from the executive we used as the starting point of our implementation, so a fully-fledged hypervisor would still be less than 6 kLOC. This compares to 9 kLOC kernel plus 27 kLOC user-mode code in NOVA on x86 [SK10].

## 6.2 Performance

Given the lack of a hardware implementation of the architecture, or at least a cycle-accurate simulator, no real performance evaluation is possible. However, we can get very rough estimates by collecting traces and weighting instructions with their known or estimated latencies. We use the estimated latencies shown in Table 1, which are appropriate for an ARM A9 core and a typical memory system.

The most uncertain of these latencies is the cost of entering hyp mode, estimated at 50 cycles. This may look incredibly optimistic to those used to x86, where VM exits are more than an order of magnitude more expensive. Remember that ARM is a RISC architecture; kernel entry/exit costs of the order of ten cycles, compared to many hundreds on x86. Also, the ISA avoids inherently-expensive operations, such as automatic saving of guest state (see Section 3.3).

For a number of microbenchmarks we measured instruction counts and converted them into approximate cycle counts under the above assumptions. The result is shown in Table 2, where *instr.* refers to the instruction count extracted form the simulation and *cycles* is the resulting estimate of execution time in cycles. Note that even if our estimate for a hypervisor trap was off by a factor of four (which seems extreme, given our experience with the architecture), this would add 150 cycles to each of the entries in the table, and would not change the picture significantly.

The hypervisor entry/exit costs are the estimated mode-switch costs plus the software cost of saving/restoring enough state to execute C code in the hypervisor. *Hypervisor entry* is the entry cost for a hypercall, while *IRQ entry* is the cost of entering hyp mode via an interrupt vector. The comparable x86 figure (NOVA) is 4,000 cycles. *Page fault* is the cost of handling the case where the guest faults on a (guest-physical) page which has not yet been mapped by the hypervisor.

The *device emulation* figures refer to trapping and emulating access to a device register, and *acc* refers to the case where this is accelerated by hardware support for emulation. The results show that device emulation is expensive even with emulation support, and production systems will likely continue to use para-virtualization of device drivers.

*World switch* refers to switching VM context, using ARM's multi-register operations for efficiently saving and restoring state. Much of this state is kept in co-processor (MMU) or core-external (virtual interrupt controller, devices) registers which are more expensive to access than internal registers. *Lazy FPU switch* refers to the cost of switching the

**Table 2: Microbenchmarks: Estimated overheads**

| Operation | Instr. | Est. Cycles |
|---|---|---|
| Hypervisor entry | 89 | 450 |
| IRQ entry | 239 | 700 |
| Hypervisor exit | 31 | 200 |
| Page fault | 356 | 1500 |
| Device emulation | 249 | 1040 |
| Device emulation (acc) | 176 | 740 |
| World switch | 2824 | 7555 |
| Lazy FPU switch | 127 | 950 |

FPU lazily, i.e. when a guest application attempts to access it.

# 7. EXPERIENCE

We found that the ARM virtualization extensions significantly reduce complexity of a hypervisor (compared to para-virtualization) and are also likely to reduce virtualization overheads.

Compared to x86 there is some added complexity, for example saving and restoring VM state in software, but this is minor (about 200 LOC). The advantage of the ARM model is that the hypervisor can optimise the handling of VM state. For example, if an interrupt is received which is destined for the presently running VM, no state needs to be switched beyond what is needed to run the hypervisor, while x86 would save and restore redundant state in this case. The downside is that a full world switch seems to be more expensive than on x86. However, this cost could be reduced if there was a single (pipelined) instruction for saving the complete MMU (CP15) context, a saving which could be of the order of 1000+ cycles.

IRQ handling is also simple and fast on ARM, as the hypervisor can inject interrupts "blindly", while on x86 the hypervisor needs to check masks and guest priorities, and modifications of masks by the guest trap into the hypervisor. One drawback of the ARM model is that interrupt handling is configured all-or-nothing. Unless all IRQs can be handled by the guest (which seems an unusual situation), the hypervisor must be invoked on each IRQ and forward it to the respective guest. Per-IRQ configuration would be a significant improvement (shaving an estimated 700 cycles off IRQ processing), even though it would slightly increase the (anyway high) cost of a world switch (for saving/restoring the IRQ bitmap).

Instruction-emulation support is one of the strengths of the ARM model, which dramatically reduces hypervisor complexity and significantly improves performance. The exception are write-back instructions, for which no support is available. Given that the instruction has already been loaded into the core, having the hardware save it into a dedicated hypervisor register could reduce emulation cost by an estimated 250 cycles. However, this may not add much to overall performance, as emulating write-back instructions seems a rare event.

In fact, given the thorough virtualization of the processor hardware, including the MMU, instruction emulation is mostly needed for virtualising devices. However, pure virtualization of device-register accesses is likely very expensive even where the architecture fully supports emulation. We therefore expect that drivers for most shared devices will continue to be para-virtualized, making the lack of emulation support for write-back even less critical.

# 8. CONCLUSIONS AND FUTURE WORK

In the course of this project we encountered numerous simulator bugs which were new to ARM (but confirmed as bugs). This is strong indication that no-one had exercised the simulator as we have, and most likely means that we were the first to produce a fully-functional hypervisor for hardware-supported virtualization on ARM able to run unmodified Linux guests.

We found that while the ARM extensions superficially look very similar to those of the x86 architecture, the RISC nature of ARM allows for a much simpler implementation. Especially the support for instruction emulation benefits from the RISC design, and allows all required emulations to be done with very little code. While our prototype still lacks a few features, most importantly VM priorities and support for multicores, it is clear that a fully-functional ARM hypervisor can be implemented in around 6,000 LOC, vastly smaller than on x86. Work on turning the prototype into a commercial product is under way.

## Acknowledgements

We would like to thank the OK Labs engineering team for their support, especially Carl van Schaik for sharing his thorough knowledge of the ARM architecture. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

# 9. REFERENCES

[AA06] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *12th ASPLOS*, San Jose, California, USA, Oct 2006.

[ARM10] ARM Architecture Group. *Virtualization Extensions Architecture Specification*, 2010. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b_virtualization_extns/index.html.

[ARM11] CoreLink system controllers for AMBA. http://www.arm.com/products/system-ip/controllers/index.php, 2011.

[Bha09] N. Bhatia. Performance evaluation of Intel EPT hardware assist. Technical report, VMWare, 2009.

[FO06] J. Fisher-Ogden. Hardware support for efficient virtualization. Technical report, University of California, San Diego, 2006.

[Gree10] INTEGRITY Secure Virtualization. http://www.ghs.com/products/rtos/integrity_virtualization.html, May 2010.

[Hei08] G. Heiser. The role of virtualization in embedded systems. In *1st WS Isolation & Integration Emb. Syst.*, pages 11–16, Glasgow, UK, Apr 2008. ACM SIGOPS.

[Hei09] G. Heiser. Hypervisors for consumer electronics. In *6th IEEE Consumer Comm. & Networking Conf.*, pages 1–5, Las Vegas, NV, USA, Jan 2009.

[HL10] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *1st APSys*, pages 19–24, New Delhi, India, Aug 2010.

[HSH+08] J.-Y. Hwang, S.-b. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer Comm. & Networking Conf.*, pages 257–261, Las Vegas, NV, USA, Jan 2008.

[Kro09] K. L. Kroeker. The evolution of virtualization. *CACM*, 52(3):18–20, 2009.

[NSL+06] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology J.*, 10(3), Aug 2006.

[OKL11] Open Kernel Labs Website. http://www.ok-labs.com/about/about-ok-labs, Jan 2011.

[RedB10] Red Bend Software website. http://www.redbend.com/, Dec 2010.

[SK10] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *5th EuroSys Conf.*, Paris, France, Apr 2010.

[VMwa10] VMware mobile virtualization platform website. http://www.vmware.com/products/mobile/, Dec 2010.

# Traveling Forward in Time to Newer Operating Systems using ShadowReboot

Hiroshi Yamada and Kenji Kono
Keio University, JST CREST
3-14-1, Hiyoshi, Kohoku-ku, Yokohama, Japan
yamada@sslab.ics.keio.ac.jp, kono@ics.keio.ac.jp

## ABSTRACT

This paper presents *ShadowReboot*, a virtual machine monitor (VMM)-based approach that shortens the downtime for software updates during an OS reboot. ShadowReboot reboots the guest OS in the background by spawning a VM dedicated to an OS reboot and enables the user to switch over to the rebooted state where the updated kernel and applications are ready for use. ShadowReboot provides an illusion to the users that the guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use. ShadowReboot offers the following advantages. It can be applied to any patch to the kernels and even system configuration updates. Second, it does not need any special patch requiring intimate knowledge about the target kernels. Third, it does not require any target kernel modification. We implemented a prototype in VirtualBox 3.0.8 OSE. Our preliminary experimental results show that ShadowReboot shortened the downtime of commodity OS reboots on Windows XP and five Linux distributions (Gentoo, Fedora, Cent, Ubuntu, and SUSE) by 43 to 96%.

## 1. INTRODUCTION

Operating system (OS) reboots are an essential part of updating contemporary kernels and applications on laptops and desktop PCs. The downtime during OS reboots severely disrupts the users' computational activities. While the OS is rebooting, the user cannot use his or her PC. This disruptive downtime is getting longer and more costly since there are more and more software updates. This long disruption caused by these OS reboots discourages users from conducting them, failing to enforce them to conduct software updates. Although announced updates should be applied as soon as possible because they tend to include fixing critical vulnerabilities, the resultant downtime may force users to delay updating their software. As a result, users cannot enjoy the new functionality or a better performance, and even worse, the unfixed vulnerabilities can be exploited by attackers. Research literature [1] notes "many desktop machines are not rebooted to apply kernel patches because of the burden imposed by rebooting".

To eliminate the need for an OS reboot with software updates, dynamic updatable kernels are a powerful way to apply patches to the kernels at runtime. However, making the systems "reboot-free" is still difficult even when using dynamic updatable kernels for the following reasons. First, existing dynamic updatable kernels are often designed for fixing bugs in the kernel code region, such as condition misses [1]. Therefore, it is difficult to manage the semantic changes to memory objects, such as when adding a new field to a data structure. They also cannot manage system configuration updates because a restart of all the processes is not involved. In these cases, we have no choice but to conduct an OS reboot. Second, some dynamic updatable kernels need intimate knowledge about the target kernels [3, 7]. To use them, we have to develop special patches from the original ones. This task is non-trivial because it requires knowledge about the internal structures of the target kernels at the source code level. Lastly, we have to pay a high engineering cost for redesigning and modifying a large part of the kernels [4, 2, 8]. This is not easy because recent kernels are more complex, and some of them are closed-source and/or proprietary.

This paper presents *ShadowReboot*, which shortens the downtime of OS reboots during software updates. ShadowReboot is designed to avoid the weak points of dynamic updatable kernels. First, ShadowReboot can be used to apply any patch to the kernels and can even be used for system configuration updates. Second, ShadowReboot does not need intimate knowledge about the target kernels at the source code level; we do not have to develop kernel modules or special patches. Finally, ShadowReboot requires no modification of the target kernels.

In ShadowReboot, we exploit the file access patterns of commodity OSes during their reboots in software updates. There are two key observations behind ShadowReboot. One is that commodity OSes during reboots tend to access files in administrative directories in which the system configuration files and shared components files are stored; these files are basically unmodified by the non-administrative tasks, such as web browsing and e-mailing. The other is that almost all the files in the user directories are not accessed during the OS reboots. These observations bring us to the following point; even if we heavily modify the files in the user directories while the OS is simultaneously rebooting, the modification does not interfere with the reboot activity and vice versa. This motivates us to parallelly run the users' non-administrative tasks and an OS reboot.

To execute the users' applications and an OS reboot in parallel, ShadowReboot spawns a VM dedicated to an OS reboot, which is called a *reboot-dedicated VM*. Since the OS is rebooted on the reboot-dedicated VM, the user can continue to execute applications on the original VM. ShadowReboot restores a snapshot of the reboot-dedicated VM where the reboot is completed, keeping the disk consistency between the original and reboot-dedicated VM. Although the user still has to deal with the restoration downtime, it is shorter than that of directly rebooting an OS on the original VM. Through these operations, ShadowReboot provides an illusion to users that a guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use.

To create a rebooted state that is consistent with users' operations, we introduce the notion of *reboot-terms*, where users can modify their working directories specified in advance while not modifying their administrative directories. We need to pay close attention to restoring the created rebooted state. Since we restore the snapshot of the reboot-dedicated VM, the restored VM naturally provides users with only the disk states of the reboot-dedicated VM. This means that the users' activities saved in the original VM are discarded. To solve this problem, ShadowReboot performs the following operation. It starts a reboot-term on the original VM when the reboot-dedicated VM is spawned. Next, when the rebooted state is restored, ShadowReboot maintains the disk states of user directories on the original VM by using an unrollback virtual disk whose state is not affected by the restore operation. By doing so, we can retain the saved users activities and make the states of the running processes, such as the daemons, consistent with the administrative files.

We implemented a prototype in VirtualBox 3.0.8 OSE. Our preliminary experimental results show that ShadowReboot shortened the downtime of commodity OS reboots on Windows XP and five Linux distributions (Gentoo, Fedora, Cent, Ubuntu, and SUSE) by 43 to 96%.

## 2. KEY OBSERVATIONS

In ShadowReboot, we exploit the file access patterns of commodity OSes during their reboots in software updates. We checked the directories and files accessed during the OS reboots after software updates. To obtain the names, we started monitoring the file accesses when an OS shutdown operation is triggered after a software update is completed. We continued to monitor the file accesses until the OS displays a login prompt. We ran Windows XP professional edition (`winxp`) and five Linux distributions, Fedora Core 10 (`fedora`), Ubuntu 9.04 (`ubuntu`), Gentoo Linux 2007.0 (`gentoo`), CentOS 5.3 (`cent`), and OpenSUSE (`suse`). Their configurations are in default. The updates conducted on `winxp` include all the Windows updates for the service pack 3 that need reboots, which were announced before October 2010, and an Internet Explorer upgrade to version 8. For the five Linux distributions, we applied a kernel patch to each kernel.

The results on `winxp` show that it basically accesses the same files and directories during the reboots. It frequently accesses \WINDOWS\system32\ and \WINDOWS\ Fonts\ for restarting services. In the shutdown phases, `winxp` accesses \Program Files\ to stop applications. Since the Windows Updates request an OS reboot during logging on, the shutdown phases involve accessing the user setting files such as \Document and Settings\username\NTLOGIN.DAT, and \Document and Settings\username\NTLOGIN.LOG. `Winxp` also stores the volume states in \System Volume Information\ for recovery. In the boot phases, winlogon.exe accesses \Documents and Settings, \Documents and Settings\NetworkService and \Documents and Settings\LocalServices for a logon. `Winxp` also sometimes accesses \WINDOWS\SoftwareDistribution\ and \WINDOWS\LastGood.Tmp for unknown reasons.

The results from the five Linux show that during each reboot all of them access the administrative files, but never access the user files in /home. All the Linux distributions frequently access files in /lib in their boot phase because almost all the daemon processes are linked to the glibc shared library whose files are found in /lib/. In addtion, the files in /etc are often accessed because the configuration files are conventionally stored in /etc. Each Linux distribution conducts slightly different file accesses due to the difference in configurations of the daemon processes. For example, `fedora` accesses /lib/libselinux.so, while `gentoo` does not. This is because `gentoo` does not support the selinux service that `fedora` does.

These results indicate that the commodity OSes during their reboots access specific files and directories. They tend to access files in administrative directories that cannot be modified without administrative privileges. On the other hand, almost all the files in user directories, such as \Documents and Settings\username\My Documents and /home/users/ Desktop, are not accessed. The characteristics of the file access patterns bring us to the following point; even if we heavily modify the files in the user directories while the OS is simultaneously rebooting, the modification does not interfere with the reboot activity and vice versa. For example, even if we run non-administrative tasks and an OS reboot in parallel with a shared disk, the tasks' activities do not interfere with the OS reboot activity. This motivates us to execute the users' tasks and an OS reboot in parallel.

## 3. SHADOWREBOOT

ShadowReboot makes use of a system virtualization to parallelly execute the users' applications and an OS reboot. ShadowReboot conducts an OS reboot in the background by spawning a VM dedicated to an OS reboot, which is appropriately called a *reboot-dedicated VM*. Since the OS is rebooted on the reboot-dedicated VM, the user can continue to execute applications on the original VM. After the OS reboot is complete, ShadowReboot takes a snapshot of the reboot-dedicated VM. It enables the user to restore the snapshot states at their convenience. Although the user experiences some downtime during the restoration, it is shorter than that of directly rebooting an OS on the original VM.

We also have to focus on restoring a snapshot from the reboot-dedicated VM. ShadowReboot runs a pair of VMs: the original and reboot-dedicated one. Since each VM has its own file system and individual disk states, the file updates of the two VMs are reflected on each virtual disk. During shadow rebooting, the users' applications may issue file writes to store their data on the virtual disk of the original VM, while the files may be modified in the reboot-dedicated VM. Since the state of the reboot-dedicated VM is restored, the files updates issued on the original VM are discarded. As a result, the user's tasks may roll back to the point when

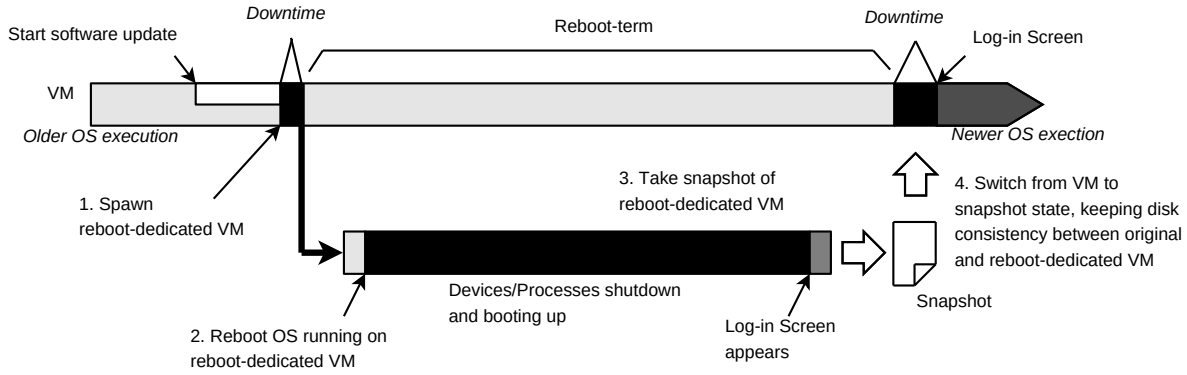


**Figure 1: Overview of ShadowReboot.**

the reboot-dedicated VM was spawned. If the VM continues to use the disks from the original VM after the restoration, the disk contents updated on the reboot-dedicated VM are discarded.

To successfully build a rebooted state that is consistent with the users' operations, we introduce the notion of *reboot-terms* during which the users can modify their working directories specified in advance while not modifying the administrative directories. ShadowReboot performs the following operation, exploiting the reboot-term. ShadowReboot starts a reboot-term on the original VM when the reboot-dedicated VM is spawned, and finishes it when the snapshot restoration is complete. Although the users activities on the original VM is limited since they cannot modify the administrative directories, they can do non-administrative tasks such as web browsing and e-mailing. Next, when we restore the rebooted state, ShadowReboot keeps the directories specified as working directories on the original VM and restores the other directories on the reboot-dedicated VM. By doing so, ShadowReboot allows users to access the files that the running processes, such as daemons, are based on. This means that the constructed processes' states are consistent with the files on the disks.

An overview of ShadowReboot is shown in Figure 1. An OS is rebooted on the reboot-dedicated VM after the software updates are applied. When the reboot-dedicated VM is spawned, ShadowReboot starts a reboot-term. After that, we restore the directories specified as working directories on the original VM and the other directories on the reboot-dedicated VM. Through these operations, ShadowReboot provides the users the illusion that a guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use.

# 4. DESIGN

Several questions are posed while designing ShadowReboot, such as (1) how can we efficiently spawn a reboot-dedicated VM, (2) how can we appropriately restore directories from the original and reboot-dedicated VM, (3) how do we check whether or not ShadowReboot successfully create the rebooted state. We answer them in this section.

## 4.1 VM Fork

We need an efficient way to create a reboot-dedicated VM. A naive approach to creating a reboot-dedicated VM is to run a new VM instance with the same configuration as the original VM. However, at every announcement of a software update, we have to create a new VM instance, copy the image of the VM, boot an OS, perform the software update, and conduct an OS reboot. This is tedious, and thus, may fail to encourage users to update their software.

To efficiently create a reboot-dedicated VM, we introduce a *VM fork* that forks a running VM, borrowing an idea from the existing literature [5, 9]. The semantics of the VM fork are similar to those of the familiar process fork; users issue a fork call to the VMM that creates a child VM. The child VM inherits the runtime state of the parent VM such as memory and registers. In addition, it proceeds with an identical view of the system. The child VM has its own independent copy of the OS, virtual disk, network interface card (NIC), and snapshot. The state updates of the child VM are not propagated to the parent.

To reclaim the memory pages for a child VM execution, we make use of a page sharing mechanism running inside the VMM to produce behavior like memory ballooning. This page sharing mechanism allows one physical page to be shared with several virtual pages whose contents are the same. If there are not enough memory pages to run a reboot-dedicated VM, we run a process on the original VM that fills its memory region with the same data. The page sharing mechanism reclaims the memory pages of the process, which means the number of free memory pages increased. By doing so, we can reclaim the memory pages for a child VM without needing kernel modules such as a balloon driver.

## 4.2 Unrollback Virtual Disk

To keep the disk updates on the original VM in restoring the rebooted state of reboot-dedicated VM, we make use of an unrollback virtual disk that is independent of a snapshot restoration function. Unlike normal virtual disks, unrollback virtual disks do not roll back even if the VM is restored to a snapshot. By leveraging the unrollback virtual disks, we can keep the files and directories on the original VM after the restoration. While the guest OS is rebooting on the reboot-dedicated VM, we save the computational activities into the file system on the unrollback virtual disks. After the original VM has been restored to a snapshot of the reboot-dedicated VM, we can access the saved contents by mounting the target partitions in the unrollback virtual disks connected to the original VM since the state of the unrollback virtual disk is not affected by the restore operation.

A typical system configuration of Linux systems is that the mount point of the working directory (/home/users/work) is assigned to the unrollback virtual disk and the other directories' mount points are assigned to standard virtual disks. We

shadow-reboot the VM after updating the software. Daemon processes become available for use based on their configuration files put on the administrative directories such as /etc on the reboot-dedicated VM, while we execute our computational tasks on the original VM, such as web browsing, e-mailing, and word processing. When we restore the rebooted state of the reboot-dedicated VM, the /home/users/work directory is not restored because its mount point is assigned to the unrollback virtual disk. As a result, the built VM provides the /home/users/work directory of the original VM and the other directories of the reboot-dedicated VM. This indicates that we can successfully preserve the disk updates in the user directories on the original VM and keep the daemons states that are consistent with the files in the administrative directories.

In a way that is similar to that in unrollback virtual disks, some approaches can protect the files and directories from snapshot restoring. We can protect them by using an additional VM on which an NFS server is running. The files and directories put on the NFS server are not affected by the snapshot restoration. However, in this approach, we have to set up a VM and experience network virtualization overhead that tends to cause a large performance penalty. We can also protect the files and directories by sharing them with the host OS. Although they are not rolled back by the snapshot restoration, the users sometimes want isolation between the VMs and the host to protect the host against VMs compromised by viruses or attackers.

### 4.3 File Access Monitor

It is helpful to prepare a mechanism that checks whether ShadowReboot successfully creates a rebooted state. Since administrative directories are restored from the reboot-dedicated VM, ShadowReboot naturally cancels any updates to the directories on the original VM during the OS reboots on the reboot-dedicated VM. On the other hand, ShadowReboot also discards the updates to working directories on the reboot-dedicated VM because they are restored from the original VM. If we do not detect any updates that violate the constraints of the reboot-terms, we fail to systematically create a consistent rebooted state; on the restored VM, the running processes' states are inconsistent with the files in the disk and/or the updates to the working directories are discarded.

To check whether ShadowReboot successfully creates a rebooted state, we prepare two processes. One monitors the access to the working directories on the reboot-dedicated VM. The other monitors the access to the administrative directories on the original VM. We can implement such processes by using a file monitoring mechanism such as a filter driver or i-notify. When the update is detected on either VM, the process tells it the user and recommends to conduct a normal OS reboot. We are now implementing this feature on Linux and Windows.

## 5. PRELIMINARY EXPERIMENTS

We are implementing a prototype in VirtualBox 3.0.8 OSE. We performed a preliminary experiment to examine the basic performance of ShadowReboot. The experiments described in this section are conducted on a DELL OptiPlex 780DT with a 3 GHz Core 2 Duo processor with 4 G of memory and a 160 GB SATA disk. Our prototype is running on this machine, where Linux 2.6.34 is also running.

To confirm ShadowReboot successfully manages the downtime of OS reboot, we compared the downtime of ShadowReboot and normal OS reboots. Our prototype causes downtime at two points. One point is when a VM fork is invoked and the other is when a snapshot of a rebooted state is restored. We measured the downtime caused by VM forks and snapshot restorations. We regard the sum of the two downtime as the ShadowReboot downtime. We used six commodity OSes (`fedora`, `ubuntu`, `gentoo`, `cent`, `suse`, and `winxp`) as our guest OSes, which were described in Section 2. Each VM is assigned one VCPU and is connected to a 20 GB normal virtual disk and a 10 GB unrollback virtual disk as a primary master and slave respectively. We varied the VM memory size to 256, 512, 1024, 2048 and 2560 MB. The maximum memory size the VirtualBox can assign on our environment is 2560 MB. The measurement was performed using the five Linux distributions and `winxp`.

Table 1 lists the downtime of ShadowReboot ($SR$) and normal OS reboots ($NR$). The results show that the downtime of ShadowReboot is shorter than that of normal OS reboots. For example, the downtime of ShadowReboot at 256 MB is 96.6% shorter than that of the normal OS reboot in `cent`. Even in `winxp`, the downtime of ShadowReboot is 71.9% shorter than that of the normal OS reboot. When we used 2560 MB of memory, the downtime of ShadowReboot is 1.97 seconds in `gentoo`, while that of the normal OS reboot is 58.21 seconds. Although the ShadowReboot downtime is about 10 seconds in `winxp`, it is 43.4% shorter than that of the normal OS reboot.

Table 1 also lists the downtime of VM forks and the restoring snapshots of the reboot-dedicated VMs. The downtime of VM forks is different in these cases. Since our prototype simply uses the snapshot functionality to fork a VM, the downtime is equal to the downtime of taking snapshots. In VirtualBox, longer downtime when taking snapshots is incurred since it saves all the physical pages allocated by the VMM. For example, the VM fork in `gentoo` stops the VM for 0.55 seconds even when the VM is assigned 2560 MB. This is because `gentoo` does not aggressively utilize the memory, just after a log-in. On the other hand, the VM fork downtime in `winxp` is 8.10 seconds at 2560 MB. `Winxp` accesses all the pages due to its mysterious behavior, which forces the VirtualBox to assign the VM memory pages.

The downtime of restoring a rebooted state also tends to be stable even if the memory size is varied, except for `cent` and `suse`. In VirtualBox, the downtime of restoring a snapshot depends on how much memory a guest OS uses. In `cent` and `suse`, their daemons use the memory in their boot phase, taking the memory size of the machine into consideration. For example, `readahead_early` warms the file cache by accessing the files that are frequently used. Although Windows is equipped with such a feature, we were able to take a snapshot before it runs.

## 6. RELATED WORK

Using dynamic updatable kernels is an effective way to apply patches to the kernels at runtime so that we do not need to conduct an OS reboot [1, 7, 3, 4, 2, 8]. Ksplice [1] dynamically translates the function code at a safe time when no thread's instruction pointer falls within that function's text and when no thread's kernel stack contains a return address within that function's text. Ksplice is designed to manipulate the text region, not to handle the memory objects in the

**Table 1: Downtime of ShadowReboot and normal OS reboot.**

| Memory size | fedora (second) | | | | ubuntu (second) | | | | gentoo (second) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SR | | | NR | SR | | | NR | SR | | | NR |
| | VM fork | Restore | Total | | VM fork | Restore | Total | | VM fork | Restore | Total | |
| 256 MB | 2.19 | 2.49 | 4.68 | 42.23 | 2.16 | 2.43 | 4.59 | 27.47 | 0.42 | 1.38 | 1.90 | 55.39 |
| 512 MB | 2.52 | 2.56 | 5.08 | 42.80 | 2.38 | 2.41 | 4.79 | 27.63 | 0.47 | 1.43 | 1.90 | 54.89 |
| 1024 MB | 2.61 | 2.38 | 4.99 | 44.55 | 2.36 | 2.46 | 4.82 | 39.61 | 0.48 | 1.38 | 1.86 | 57.82 |
| 2048 MB | 2.73 | 2.37 | 5.10 | 45.03 | 2.71 | 2.53 | 5.12 | 43.64 | 0.53 | 1.43 | 1.96 | 58.17 |
| 2560 MB | 2.74 | 2.39 | 5.03 | 45.04 | 5.27 | 2.78 | 2.49 | 45.49 | 0.55 | 1.42 | 1.97 | 58.21 |

| cent (second) | | | | suse (second) | | | | winxp (second) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SR | | | NR | SR | | | NR | SR | | | NR |
| VM fork | Restore | Total | | VM fork | Restore | Total | | VM fork | Restore | Total | |
| 2.32 | 2.42 | 4.74 | 141.11 | 2.26 | 3.16 | 5.42 | 30.95 | 1.60 | 2.30 | 3.90 | 13.88 |
| 3.72 | 3.32 | 7.04 | 154.09 | 3.71 | 4.86 | 8.57 | 30.71 | 2.39 | 2.38 | 4.77 | 13.73 |
| 3.75 | 3.20 | 6.95 | 132.84 | 4.11 | 5.12 | 9.23 | 43.29 | 6.28 | 3.83 | 2.45 | 16.67 |
| 3.79 | 3.15 | 6.94 | 142.83 | 4.00 | 5.29 | 9.29 | 56.24 | 6.67 | 2.32 | 8.99 | 18.42 |
| 3.86 | 3.35 | 7.21 | 132.05 | 4.46 | 5.22 | 9.68 | 55.99 | 10.41 | 8.10 | 2.31 | 18.38 |

kernel heap region. Additionally, this approach cannot update the *non-quiescent* kernel functions that are always on the call stack of some kernel threads. These approaches also do not manage the system configuration changes and shared component updates because the running processes are not restarted. We can complementarily use ShadowReboot to handle such updates with shorter downtime.

Some approaches require the development of special patches from the original ones. LUCOS [3] forces users to implement new functions that can handle the kernel memory objects to keep them consistent before and after the translation. In DynAMOS [7], users have to investigate how the target functions are used by the kernel threads and implement a routine that consistently updates them. ShadowReboot does not need to perform such tedious tasks.

There are approaches that involve paying the high engineering cost of redesigning and modifying a large part of the kernels. To use K42's techniques [4, 2, 8] on commodity OS kernels, we have to redesign the target kernels in an object-oriented manner. Modifying commodity OS kernels is often difficult because recent kernels are complex and some of them are closed-source and/or proprietary. ShadowReboot does not require any modification of the OS kernels.

MicroVisor [6] conducts a process migration between two VMs connected to a shared network storage, such as an NFS server and a SAN. An administrator runs the applications in one VM and maintains the kernel in the other. When the maintenance has finished, the applications running on the older kernel in the first VM are migrated to the newer kernel in the second VM. Finally, the first VM is discarded. Although these approaches successfully hide the downtime of the kernel maintenance, the process migration is unsuitable for system configuration changes and shared components updates. Since migrated processes are running with the configuration of the older OS, their states remain older on the newer OS. An administrator has to carefully choose the processes that can be migrated to avoid a configuration mismatch of the processes between the older and newer OSes, based on which configuration or component is updated. ShadowReboot systematically provides users a consistent system state by introducing the reboot-terms.

# 7. CONCLUSION AND FUTURE WORK

This paper described ShadowReboot, a VMM-based approach that shortens the downtime of OS reboots for software updates. ShadowReboot provides the illusion that a guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use.

We need to implement a file access monitor to guarantee that ShadowReboot successfully creates a rebooted state. After that, we conduct experiments to confirm that ShadowReboot can successfully update the commodity OSes with shorter downtime. We also explore ways to schedule a reboot-dedicated VM to prevent it from severely interfering with the users computational tasks on the original VM.

# 8. REFERENCES


[1] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the 4th ACM European Conf. on Computer Systems*, Apr. 2009.

[2] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Technical Conf.*, Jun. 2007.

[3] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live Updating Operating Systems Using Virtualization. In *Proc. of the 2nd ACM Int'l Conf. on Virtual Execution Environments*, Jun. 2006.

[4] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *Proc. of the 1st ACM European Conf. on Computer Systems*, Apr. 2006.

[5] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for could computing. In *Proc. of the 4th ACM European Conf. on Computer Systems*, Apr. 2009.

[6] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of the 11th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[7] K. Makris and K. D. Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proc. of the 2nd ACM European Conf. on Computer Systems*, Mar. 2007.

[8] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration. In *Proc. of the USENIX Annual Technical Conf.*, Jun. 2003.

[9] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, Oct. 2005.

# Vis: Virtualization Enhanced Live Acquisition for Native System

Miao Yu, Qian Lin, Bingyu Li, Zhengwei Qi, Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiao Tong University
{ superymk, linqian, justasmallfish, qizhwei, hbguan } @ sjtu.edu.cn

## Abstract

Focusing on obtaining in-memory evidence, current live acquisition efforts either fail to provide accurate native system physical memory acquisition at the given time point or require suspending the machine and altering the execution environment drastically. To address this issue, we propose Vis, a light-weight virtualization approach to provide accurate retrieving of physical memory content while preserving the execution of target system. Vis encapsulates the native system into a single virtual machine and then conducts accurate acquisition by manipulating nested page table in hypervisor. We present the design and implementation of Vis, prove its acquisition reliability and evaluate its performance in live acquisition scenarios.

## 1. INTRODUCTION

A typical computer forensics scenario has three steps: acquisition, analyzing and reporting [18]. Focusing on the stages of acquisition and analyzing, computer forensics proposes two key challenges: how to obtain the complete system state and how to analyze the retrieved image effectively. The former one is more important because missing evidence leads to an incomplete or wrong investigation result, even with an incomparable analyzing technology.

Transcending static acquisition strategies, live acquisition extends the information gathering range of forensics examiner, i.e., involving the volatile data. Considering criminal evidence being stored on permanent I/O device only [5], most static acquisition tools clone disk offline to accurately obtain the evidence. Nevertheless, evidence data existing in volatile memory without disk correspondence are totally beyond the acquisition scope of static acquisition tools. To address such issue, the requirement of live acquisition becomes essential. Live acquisition tools can extract the volatile data in the memory of the target system without blocking it. These data involve process information [4], process list [9], kernel objects and raw memory content [3], which may be leveraged to record and reproduce the criminal scene.



Based on the architecture difference, previous software live acquisition solutions can be divided into two categories. The first one is *Virtualization Introspection*, which means the target system is wrapped in a Virtual Machine (VM) while the acquisition module exists in a hypervisor like Xen. VIX tools [9], Ruo's work [3], Srinivas's work [10] and BodySnatcher [16] all belong to this type. The second one is *Non-Virtualization Introspection*. It is designed to obtain indicated volatile system state with a minimal environment impact. Iain et al. [17] list several practical tools for different scenarios, including Win32dd, KnTTools and Fport. Memoryze [11] is another popular user process forensic tool of this type.

While owning the ability of unearthing tremendous volume of volatile data, live acquisition also faces significant challenges and risks. The first challenge is that previous virtualization based live acquisition methods alter the system environment significantly. The reason comes from the fact that many previous approaches required loading hypervisor prior to the launching of operating system (OS) [3, 9, 10]. When employing this method on a non-virtualized host, the forensic examiners more or less change the system running environment. In the extreme case, rebooting even reinstalling the whole system is required, thus causing a great loss of information from volatile memory. The second challenge raises from the fact that the system is not static [2]. Contents in physical memory changes with the running processes, making those previous In-OS live acquisition methods unable to guarantee the accuracy of the retrieved physical memory content at the given time point unless suspending the machine. However, providing ideal suspending functionality would require hardware support [8]. As a result, practical In-OS live acquisition tools, like Win32dd and Memoryze, never consider result accuracy as one of their design goals. Also, the acquisition task would take longer when transmitting data over cables. It is reported that BodySnatcher [16] requires suspending the target system for 45 minutes to accomplish a complete 128MB RAM acquisition over 115kbps serial I/O cable. Moreover, it is noteworthy that dumping an accurate physical memory image is difficult by manipulating all page tables for In-OS live acquisition tools, because possible existence of hidden processes makes it tough to actively trace all working page tables.

In this paper, we propose a novel acquisition system named *Vis* to access these challenges. Leveraging virtualization approach, Vis provides accurate retrieving of native system physical memory while preserving the execution of target

system. To validate our approach, we have implemented a proof-of-concept prototype and conducted a series of evaluations. The result shows that even under high pollution rate during the acquisition period, Vis can still ensure the accuracy while preserving the target system execution. Moreover, the performance evaluation result demonstrates that Vis is able to retrieve an accurate system image in 105.86 seconds comparing with a range of 17∼76 seconds for Win32dd, 18 minutes for Hypersleuth on an 1Gbps network. Meanwhile, it incurs 9.62% performance overhead to existing applications. These results prove that Vis owns practical value in real world application.

The rest of the paper is organized as follows. Section 2 presents Vis's design model. Section 3 evaluates Vis through experiments. Finally, we conclude our work in Section 4.

# 2. DESIGN & IMPLEMENTATION

We propose two key techniques termed *Late-Virtualization* and *Virtual-Snapshot*, to fulfill the design requirement of Vis.

## 2.1 Late-Virtualization Approach

Late-Virtualization technique is used to insert a lightweight hypervisor after the target OS is started up as well as keep hypervisor functioning without suspending the target system. Late-Virtualization leverages the wide support of hardware virtualization on commercial x86 processors to fulfill the design goal. In current prototype, Vis employs Intel VT-x technology [1].

Intel VT-x separates the CPU execution into two modes: *VMX root mode* and *VMX non-root mode*. At any time point, CPU runs on only one of the two modes. Software running in hypervisor can supervise guest machine execution by pre-defining the interested event in VMCS (*Virtual Machine Control Structure*) which contains VM and hypervisor execution environment data. After that, any sensitive instruction executed in a guest VM is interrupted by a VMEXIT event and traps to the hypervisor. The control flow never returns back unless hypervisor finishes handling the event and then explicitly resumes running the guest machine.

A typical hypervisor launching consists of three steps. First, the VMX root-mode is enabled. Second, the CPU is configured to execute the hypervisor in root-mode. Third, the guests are booted in non-root mode. It worths noting that each core can run at most one hypervisor at any time in current VT-x implementation. Thus we do not consider the recursive virtualization situation due to the lack of hardware virtualization support inside guest virtual machine. Since Intel VT-x allows to startup a hypervisor at any time, we change the third step to continue running the virtualized native system in order to delay launching hypervisor. Figure 1 depicts how the native system environment changes after loading/unloading Vis. During the loading phase, Late-Virtualization builds the required virtualization environment and wraps the target OS into VM on the fly. Loaded as an OS driver, Vis shares the same usage model with many existing forensic tools listed in [17] and requires memory by invoking standard OS APIs. One potential problem is that in order to load the acquisition tool itself, certain memory space is needed and thus there is a potential of jeopardizing evidences in freed memory space. Vis meets this issue by claiming that according to the Locard's exchange

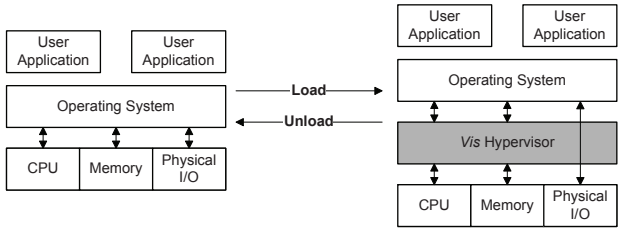


**Figure 1: The Overview of Late-Virtualization Architecture.**

principle [6], it is inevitable to bring in modification to the observed object. However, considering the accurate volatile information gathered from the remaining memory, this little modification within Vis installation is acceptable. After all, no zero invasive solution for a *posteriori* forensic analysis exists [12].

One fundamental assumption of Vis is the need for a trustworthy hypervisor. This is shared by many previous research efforts [7, 14, 15]. Even for late-launched hypervisors, previous studies also make the same assumption [12, 16]. Specifically, NewBluePill project [12] discusses how to resist attacks from guest OS by constructing private page table and shadowing control register accesses. Besides, [12] mentions that the hypervisor code can be attested before loading by employing Trusted Platform Module (TPM). Adopting these security approaches in Vis only requires more engineering effort, lengthens the time needed during loading and incurs slight performance overhead at runtime. In brief, after starting up from commercial OS, Vis assumes the hypervisor is safe enough for live acquisition.

## 2.2 Virtual-Snapshot Approach

Virtual-Snapshot is used to accurately capture the content in physical memory without suspending target system's normal execution. Two challenges are identified in accurately dumping the physical memory content. First, Virtual-Snapshot should be able to identify which part of physical memory content is newly generated and point out what the original content in that location is. Second, the large size of physical memory causes a long time to acquire all the content. Supposing the target system owns 2GB physical memory, it takes more than 20 seconds to obtain a complete memory dump and output it to the local disk at 100MB/s transfer speed. Previous live acquisition approaches need to suspend the machine in order to ensure the result's accuracy. Hence, the required long suspending time makes them inappropriate in stealthy live acquisition occasion.

The first problem is solved by Nested Paging mechanism in Virtual-Snapshot. Figure 2 shows how Nested Paging mechanism translates arbitrary Guest Virtual Address (GVA) into corresponding Machine Physical Address (MPA). In traditional virtualization, Nested Paging mechanism is employed to host multiple VMs on the same physical machine. As a result, a two-level translation mechanism is needed to ensure the compatibility with legacy OSes. Since modern hardware virtualization tries to eliminate the guest OS sense of the underlying hypervisor [13], the first level address translation, which turns GVA to Guest Physical Address (GPA), still employs the original guest OS page table pointed by CR3 register as the traditional way does. At

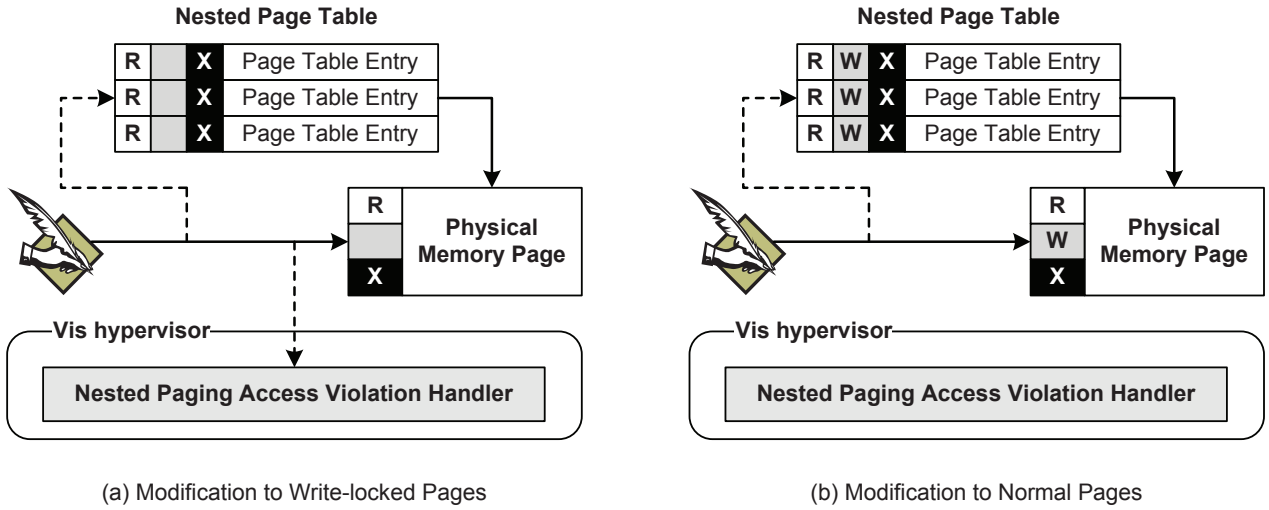


**Figure 3: Virtual-Snapshot Approach. Comparing with modification to normal pages, a different control flow is executed when modifying write locked pages.**

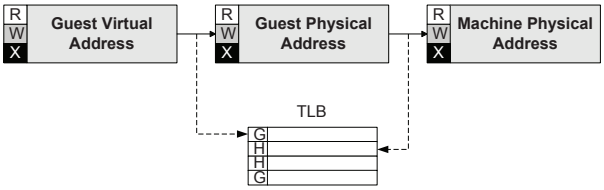


**Figure 2: Nested Paging Mechanism. This figure describes how GVA is translated into MPA. G means that the corresponding TLB entry stores GVA to MPA translation, while H means that the TLB entry is used for translating GPA into MPA. RWX stands for read, write and execute permissions on specific memory region.**

the same time, the second level address translation, which uses Nested Page Table (NPT) to translate GPA into MPA, is pointed by Nested Page Table Pointer. It is worth noting that Shadow Page Table (SPT), the traditional software Nested Paging approach which uses another sets of page tables to achieve GVA to MPA translation, can also be employed by Virtual-Snapshot in the case that hardware assisted nested paging is unsupported on legacy hardware.

In order to monitor the modification on the whole range of target system's physical memory, Virtual-Snapshot first creates an identical mapping from GPA to MPA on the second level address translation. After that, Virtual-Snapshot actively queries guest OS for its valid physical memory range by examining kernel data objects. This is because certain amount of system memory address space is reserved for existing I/O devices, e.g., graphic card, network card, etc. In addition, on x64 architecture the valid physical address space is far more tremendous than the maximum supported memory capacity currently. Hence, distinguishing physical memory range from I/O memory range and unallocated memory range helps build the acquisition range in Vis.

After obtaining the knowledge of a clear physical memory scope within Vis startup, Virtual-Snapshot revokes write permission on the whole guest physical memory range when acquisition command is issued from Vis client. As shown in Figure 3(a), achieved by manipulating the second level NPT, revoking the write permission on guest OS physical memory page forces any subsequent writing to this page to generate nested page fault before changing any single bit within it. Then, hardware automatically traps to Vis hypervisor to handle it accordingly with hardware generated guest fault frame, which includes the address of the modifying page, the allowed permission as well as the desired permission. The pre-registered nested paging access violation handler in Vis hypervisor flushes the data cache in order to get persistent view of memory, dumps the content of the trapped page, removes write lock from the trapped page by regranting the write permission, and then resumes guest machine running from the trapped instruction. Since the guest machine resumes from writing to the same guest physical page again and this time no write lock is put on the same page, the write operation succeeds without interrupting the original information flow, as shown in Figure 3(b). In this way, Virtual-Snapshot obtains the original content of the guest physical page being modified, while keeping the guest OS and application's information flow, even in the case that Vis is orthogonal to the guest machine.

The second problem is solved by an *amortized* manner of Virtual-Snapshot. According to our investigation, only a small portion of guest physical pages are modified on each processor during a single instruction execution. Hence, it is sufficient for Vis to dump only the changing pages in order to obtain a complete original content of guest physical memory. Dumping the remaining part of guest physical pages is deferred until either their modification or the end of acquisition if their content is never changed. Similar ideas were also suggested by HyperSleuth [12]. Unfortunately, comparing with our approach, HyperSleuth sufferred from degenerated performance because it ignores the memory access continuity in its algorithm. Vis improves its acquisition performance by dumping multiple continuous physical memory pages per trap. As a result, by lengthening the necessary acquisition time, Vis does not require to suspend the tar-

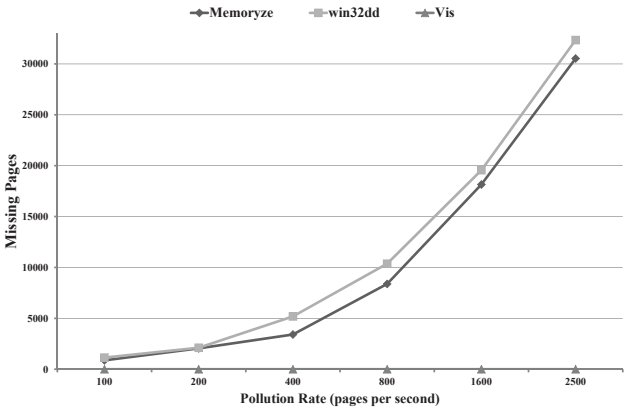


**Figure 4: Accuracy evaluation. Different page pollution rate is tested for 2GB memory dumping in each test. Missing Pages means the number of the obtained pages containing polluted content.**

get system. Furthermore, the acquisition incurred overhead is slight because Virtual-Snapshot dumps small portion of critical pages first and large part of remaining pages later in little pieces.

# 3. EVALUATION

The current Vis prototype is fully implemented on Windows 7. All experiments were conducted on a host configured with a 3.2GHz Intel i5-650 processor, 2GB RAM and a gigabit ethernet card. We use the uniprocessor x86 version of Windows 7 in our experiments. In this section, we begin with verifying the accurate live acquisition guarantees provided by Vis, then present Vis's overall performance as well as the performance impact on the target system.

## 3.1 Effectiveness

Figure 4 depicts the acquisition accuracy evaluation of Vis with another two commonly studied live forensic tools, Win32dd and Memoryze. All of these tools accomplish live acquisition without suspending the target machine. The experiment methodology is that we load the acquisition process first, and then manually start pollution process immediately after beginning acquisition. The pollution process is used to challenge the accuracy of acquisition result. This is achieved by allocating and filling memory with unique pattern of content. Thus, an ideal live acquisition tool should dump none of the polluted content. As shown in Figure 4, even in the situation that the pollution process allocates and pollutes memory at the rate of 2500 pages per second for 20 seconds, no polluted content is dumped by Vis. On the contrary, though Win32dd finishes its dumping physical memory task in 17 seconds and Memoryze, 18 seconds, they recorded 71.62% and 56.96% polluted content in the result files respectively. The result shows that Vis is able to provide accurate live acquisition.

## 3.2 Performance Impact

Table 1 presents the micro analysis which measures the overhead of handling writing CR3 and handling EPT violations, both including acquisition state and idle state. In these experiments, we perform a complete live acquisition

| | On Acquisition | Idle |
|---|---|---|
| Scenario 1: Write CR3 | | |
| #VMEXIT world switch | 966 | 548 |
| Write CR3 value | 113 | 113 |
| Handle dumping | 214934 | N/A |
| #VMResume world switch | 1355 | 760 |
| Scenario 2: Handle EPT Violation | | |
| #VMEXIT world switch | 919 | N/A |
| Clone origin page contents | 36232 | N/A |
| Reset EPT entry | 157112 | N/A |
| #VMResume world switch | 2243 | N/A |

**Table 1: Vis Micro Analysis. This figure shows the needed clock ticks in handling Write CR3 as well as EPT violation in the target system.**

compared with keeping Vis in idle state for the same time period. Hence, both scenarios have happened for tens of thousands of times during this period. Then, the average is recorded as the final result. In Table 1, "#VMEXIT World Switch" means the total clock ticks spent on hardware context switch and delegating event to the proper handler; "#VMResume World Switch" means the total clock ticks needed for both necessary cleaning work and hardware resuming VM. All the benchmarks exhibit low overhead in context switch between the target system and Vis. Besides, there is no EPT violation handling in Vis idle state because no memory write is intercepted theoretically.

We then compare the performance of Vis with legacy tools in acquisition scenario. Win32dd adopts different I/O buffer sizes to supply four speed modes. The elapsed time we recorded is directly retrieved from Win32dd acquisition report. According to our evaluation, Win32dd requires 17∼76 seconds for a complete live acquisition. In comparison, Vis is able to retrieve an accurate system image in 105.86 seconds when configured to dump 8 pages per trap.

For a complete performance evaluation, we also measure Vis in idle state runtime overhead to target system. CPU intensive and I/O intensive benchmarks were selected to run with Vis so that the quantized result can be evaluated respectively. For CPU intensive applications, we use the SPEC INT 2006 suite. For I/O intensive applications, we select IOMeter, netperf and Apache web server. In average, Vis in idle state incurs 9.62% performance impact to the target system, as presented in Table 2. Besides, the network throughput result shows it is increased by 1% in average after Vis is loaded. Such experimental result is still under investigation.

Although Vis averagely incurs 9.86% performance overhead in SPECint 2006, it is noteworthy that Vis exhibits 50.38% performance impact in MCF benchmark, which implements a graph algorithm. According to our investigation, the total EPT traverse penalty mainly contributes to this overhead. On the one hand, a high EPT Translation Look-aside Buffer (TLB) miss rate is incurred. In MCF's source code, arc_t type is 32 bytes long and nr_group variable is always set to be 870 at runtime. When executing the for-statement shown in Figure 5, the **arc** value increments 870 times 32 bytes in each for-loop. This leads to a poor space locality, which then causes higher probability in occupying EPT TLB due to the fact that TLB is shared by

| Benchmark | | Overhead |
|---|---|---|
| CPU | SPECint 2006 | 9.86% |
| I/O | IOMeter | 0.51% |
| | Netperf | -1.05% |
| | Httpd | 0.30% |

**Table 2: Vis Performance Impact**

```
165  for( ; arc < stop_arcs; arc +=
   nr_group )
166  {
167      if( arc->ident > BASIC )
168      {
169          /* red_cost =
   bea_compute_red_cost(arc); */
170          red_cost = arc->cost - arc->
   tail->potential + arc->head->potential
   ;
             ...
178      }
179  }
```

**Figure 5: A For-Statement in MCF Source Code. This for-statement is the hottest spot of TLB miss in MCF benchmark. It originally exists in the source code of pbeampp.c**

both nested paging and traditional paging in current implementation of hardware virtualization. On the other hand, every EPT TLB Miss costs the guest machine a maximum 14 times of memory access overhead to complete the nested address translation on x86 platform. This is calculated according to the following facts: traditional page table owns two level paging structures, while EPT owns four on x86 platform. Hence, the TLB Miss overhead would be greatly magnified when EPT is introduced in.

# 4. CONCLUSION

We proposed Vis, a light-weight virtualization approach to provide accurate retrieving of native system state while the target system stays running. Vis achieves its goal via two key techniques of Late-Virtualization and Virtual-Snapshot. Late-Virtualization is used to provide an isolated running environment for live acquisition tools after the target OS is started up. Virtual-Snapshot is employed to accurately obtain target system's physical memory content at the given time point. It avoids suspending target system during main memory content acquisition by adopting the amortized manner in acquisition. A proof-of-concept prototype has been developed to obtain physical memory content on Windows 7. The evaluation result demonstrates that Vis can reliably provide the intended accurate live acquisition with small performance overhead.

# 5. ACKNOWLEDGEMENT

The authors would like to appreciate the anonymous reviewers for their insightful comments on improving this paper's presentation. Also, the authors would like to thank Xiapu Luo and Nguyen Anh Quynh for useful discussions. This work is supported by National Natural Science Foundation of China (Grant No. 60873209, 60970107, 60970108, 61073151), the Key Program for Basic Research of Shanghai (Grant No. 09510701600, 10511500102, 10DZ1500200), IBM SUR Funding and IBM Research-China JP Funding.

# 6. REFERENCES

[1] *Intel 64 and IA-32 Architectures Software Developer's Manuals*. Intel Corporation, 2007.

[2] Adelstein, F. Live forensics: diagnosing your system without killing it first. *Commun. ACM 49* (February 2006), 63–66.

[3] Ando, R., Kadobayashi, Y., and Shinoda, Y. Asynchronous pseudo physical memory snapshot and forensics on paravirtualized vmm using split kernel module. In *ICISC* (2007), K.-H. Nam and G. Rhee, Eds., vol. 4817 of *Lecture Notes in Computer Science*, Springer, pp. 131–143.

[4] Buchholz, F. *Pervasive Binding of Labels to System Processes*. PhD thesis, Purdue University, 2005.

[5] Carrier, B. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[6] Chisum, W. J., and Turvey, B. E. Evidence dynamics: Locard's exchange principle & crime reconstruction. *Journal of Behavioral Profiling 1* (Janurary 2000).

[7] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (2003), The Internet Society.

[8] Hay, B., Bishop, M., and Nance, K. Live analysis: Progress and challenges. *Computing in Science and Engg. 7* (March 2009), 30–37.

[9] Hay, B., and Nance, K. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev. 42* (April 2008), 74–82.

[10] Krishnan, S., Snow, K. Z., and Monrose, F. Trail of bytes: efficient support for forensic analysis. In *ACM Conference on Computer and Communications Security* (2010), E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., ACM, pp. 50–60.

[11] MANDIANT Corporation. Memoryze. `http://www.mandiant.com/products/free_software/memoryze/`.

[12] Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. Live and trustworthy forensic analysis of commodity production systems. In *Proceedings of the 13th international conference on Recent advances in intrusion detection* (Berlin, Heidelberg, 2010), RAID'10, Springer-Verlag, pp. 297–316.

[13] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. Trustvisor: Efficient tcb reduction and attestation. In *IEEE Symposium on Security and Privacy* (2010), IEEE Computer Society, pp. 143–158.

[14] Ning, P., di Vimercati, S. D. C., and Syverson, P. F., Eds. *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007* (2007), ACM.

[15] Payne, B. D., Carbone, M., Sharif, M. I., and Lee, W. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society, pp. 233–247.

[16] Schatz, B. Bodysnatcher: Towards reliable volatile memory acquisition by software. *Digital Investigation 4*, Supplement 1 (2007), 126 – 134.

[17] Sutherland, I., Evans, J., Tryfonas, T., and Blyth, A. Acquiring volatile operating system data tools and techniques. *SIGOPS Oper. Syst. Rev. 42* (April 2008), 65–73.

[18] Wikipedia. Digital forensic process. `http://en.wikipedia.org/wiki/Digital_forensic_process`.

# Toward Under-Millisecond I/O Latency in Xen-ARM

Seehwan Yoo, Kuen-Hwan Kwak, Jae-Hyun Jo and Chuck Yoo
Korea University
{shyoo, khkwak, jhjo, hxy}@os.korea.ac.kr

## ABSTRACT

This paper addresses the I/O latency issue within Xen-ARM. Although Xen-ARM's split driver presents reliable driver isolation, it requires additional inter-VM scheduling. Consequently, the credit scheduler within Xen-ARM results in unsatisfactory I/O latency for real-time guest OS. This paper analyzes the I/O latency in Xen-ARM's interrupt path, and proposes a new scheduler to bound I/O latency. Our scheduler dynamically assigns priorities to guest OSs so that Xen-ARM ensures to schedule the most urgent task within the system. The experimental results show that Xen-ARM with our new scheduler reduces delay spikes, latency larger than 1ms, from 16% to 1% while retaining the split driver model.

## Categories and Subject Descriptors

D.4.7 [**OPERATING SYSTEM**]: Organization and Design

## General Terms

Performance

## Keywords

Virtual machine, Real-time system

## 1. INTRODUCTION

Recently, virtualization is drawing attention to embedded systems such as mobile phones or smart pads, and several hypervisors for those mobile systems are presented because virtualization systems have several advantages over traditional embedded operating systems. First of all, those hypervisors allow multiple heterogeneous operating systems to run over a single physical machine. They support both a real-time guest OS for mobile communication and a general-purpose OS for rich applications like Web browsers or office programs. In addition, virtualization protects user data from security threats, and enhances reliability by isolating an insecure or faulty domain from the others.


*APSys 2011* '11 Shanghai, China


Although embedded virtualization has diverse advantages over traditional embedded operating systems, performance issues have been continuously challenged because those systems have inherent performance requirements while having limited hardware resources. Particularly, I/O latency in a virtual machine needs to be carefully addressed since it affects the performance of a real-time guest OS. For example, over 1ms interrupt latency leads to call misses in mobile phones, which is regarded as a serious defect.

Embedded hypervisors take several different approaches in order to achieve satisfying I/O latency for a real-time guest OS over a virtual machine. One of the most representative embedded hypervisors is OKL4 microvisor [2]. The microvisor presented unique interrupt handling originated from L4 microkernel's user-level I/O driver model [6]. The microvisor achieves low latency by architectural optimizations enough to run with commercial mobile phones. Another commercial hypervisor VirtualLogix-VLX presents different I/O model [7]. Device drivers are located within the hypervisor so that the drivers can timely handle physical interrupts, and are easily shared among multiple guest OSs.

Xen [1] is another hypervisor that is popular in server systems, and recent Xen-ARM [5] presented how Xen can be adapted to mobile devices. It enhances reliability of user data by isolating a private domain from insecure domains. In addition, it is small enough to fit in a mobile platform. For I/O handling, Xen-ARM provides *split driver model*, which isolates unreliable device drivers from guest OSs. In addition, the model makes physical devices to be easily shared among guest OSs.

However, the current credit scheduler, the default scheduler of Xen-ARM, is insufficient to support a real-time guest OS due to its long I/O latency. Note that split drivers require an additional inter-VM scheduling between the driver domain and a user domain. It is known that the credit scheduler has limitations on supporting time-sensitive applications although it has additional *BOOST* priority. Two major limitations are: First, the scheduler allows multiple domains to be boosted simultaneously, so BOOST can be easily ignored. Second, a guest domain is not boosted in some cases because it primarily focuses on fairness among CPU-bound domains.

Adding another priority such as REAL_BOOST would not completely resolve the I/O latency issues because Xen-ARM has a hierarchical scheduling structure. Since the hypervisor cannot impose task scheduling within a guest OS, the hypervisor prioritizes the entire guest OS rather than a task within the guest OS. This results in increased latency because the

hypervisor cannot distinguish urgency among tasks within different guest domains. In addition, we found that physical interrupt handling can be delayed by the virtualization because a guest OS can only disable/enable virtual interrupts; and thereby the driver domain can be preempted by a user domain within the code with interrupts disabled. For example, the driver domain can be preempted by another user domain even when it handles interrupts.

So, this paper analyzes the latency in the interrupt path of Xen-ARM, and proposes new mechanisms to bound I/O latency. This paper consists of five sections. In Section2, we review the current credit scheduler within Xen-ARM. Section 3 presents how the interrupt handling can be delayed over a virtualization system. Section 4 presents our approach to guarantee I/O latency in order to run a real-time guest OS over a virtual machine. Finally, Section 5 concludes the paper.

## 2. CREDIT SCHEDULER OF XEN-ARM

The current Xen-ARM uses the credit scheduler, which is based on the WRR (Weighted Round-Robin) scheduling algorithm. The credit scheduler primarily tries to keep fairness of CPU utilization among guest domains. In the credit scheduler, a VCPU[1] has one of three priorities, UNDER, OVER, or BOOST. VCPU begins from UNDER, at which the VCPU has remaining credits, and is ready to run. Xen-ARM periodically distributes credit over all guest OSs, and debits credit as much as the guest OS consumes the physical CPU time. When a VCPU consumes all its credit, then the priority of VCPU is changed from UNDER to OVER, and the guest OS is not scheduled to run until it becomes UNDER again. Each guest OS can specify the weight so that Xen-ARM can differentiate the CPU utilization among the guest OSs.

Regarding I/O operation, the credit scheduler has additional priority called BOOST. A guest OS can be temporarily prioritized when the guest OS has pending I/O events. In Xen-ARM, BOOST is the highest priority so that the guest OS can handle I/O events in a timely manner. When a guest OS is boosted, it preempts the currently running guest OS.

## 3. I/O LATENCY IN XEN-ARM

To test I/O latency over the current Xen-ARM hypervisor, we measured interrupt handling latency at real-time domain using ping tests. Three guest OSs are running over Xen-ARM: dom0 for driver domain, dom1 for real-time I/O domain, that is a icmp receiver with light CPU load (20%), and dom2 for CPU-intensive application (100% CPU load). Then, another external server sends icmp request packets to dom1 with random interval (0∼40ms), of which average of ping interval is 20ms. The measurement is based on ARM-cortexA9 processor, and 100Mbps network device is attached via usb interface.

Figure1 shows the cumulative latency distribution for ten thousand interrupts in the two different intervals: 1) from the physical interrupt handler at Xen-ARM to network backend driver (netback) within dom0, and 2) from the netback handler to the icmp handler within dom1.

Graph in Figure1 shows two important characteristics of latency distribution within Xen-ARM. First, latency from the physical interrupt handler at hypervisor to dom0 (Xen-netback latency) is small in most cases, but the worst-case latency is significantly large. In the figure, more than 97% of icmp requests are handled within 1 ms at dom0, but rest 3% of requests are handled with very long delay up to 60ms. So, the driver domain cannot handle interrupts in a timely manner, which means I/O latency cannot be guaranteed even though dom0 serves as a real-time domain.

Second, latency from dom0 to dom1 (netback-domU latency) can be considerably large. In the figure, although more than 84% of icmp requests are handled within 1 ms at domU, more than 16% of requests are delayed more than 1ms. This is due to the inter-VM scheduling for the communication between two domains (dom0 and dom1). To support real-time, this latency has to be bounded.

In the following sections, we focus on the latencies in these two intervals.

### 3.1 Xen-netback Latency

In Xen-ARM, all physical interrupts are handled within the hypervisor, and a guest OS handles only virtual interrupts. The hypervisor delivers virtual interrupts to the designated guest OS by setting the corresponding bit in the event channel, and it finishes interrupt handling as soon as possible in order to minimize the interrupt-disabled region. Then, the hypervisor performs scheduling so that the designated domain can quickly handle interrupts. The designated domain is boosted by the credit scheduler so that the interrupts can be handled in a timely manner.

However, as we have seen in the previous section, interrupt handling at the driver domain presents large worst-case I/O latency. To guarantee I/O latency, the worst-case execution has to be time-bounded.

#### 3.1.1 *BOOST in Credit*

Although the credit scheduler supports BOOST priority, it is well known that it lacks in supporting time-sensitive applications[4, 3]. BOOST in the credit scheduler has limitations to guarantee I/O latency because boost is easily negated. Namely, multiple domains can be boosted simultaneously. This is called *multi-boost*, which negates the original boost mechanism. Since the current credit scheduler doesn't distinguish the urgency among them, the scheduler can select less urgent domain. Multi-boost is largely observed in I/O intensive systems, and it affects the Xen-netback latency.

For example, when a driver domain finishes the backend

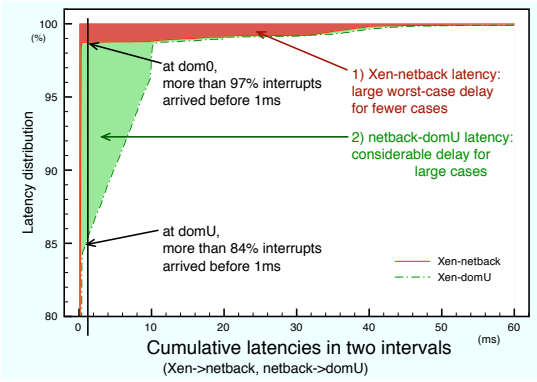


**Figure 1: Latency distribution for two intervals**

[1]In this paper, we interchangeably use VCPU, guest OS and domain if it is clear in the context.

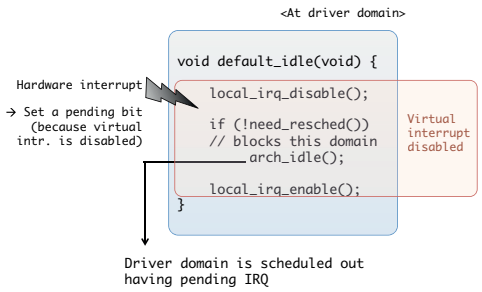


**Figure 2: Wrong idle paravirtualization**

driver handling, it awakens the designated domain, at which the VCPU is boosted. If an additional interrupt occurs, the driver domain is boosted again. So, both domains are temporarily in BOOST priority at the same time. If the hypervisor schedules the designated domain, then the interrupt handling at the driver domain can be delayed.

In addition to multi-boost, the credit scheduler doesn't boost the VCPU if the domain is OVER or the VCPU is on the run queue (we name it *non-boosted VCPU*). For example, if the I/O workload is so intensive that it leads its credit shortage, then the driver domain will not be boosted. The driver domain needs to wait until the credit is recharged, and the domain becomes UNDER again. As shown in Figure 1, this latency can vary up to multiple quantum.

### 3.1.2 Virtualized interrupt handling

Interrupt handling within a guest OS also affects I/O latency. Since interrupt handling within a guest OS is different from a native (non-virtualized) system, latency is differently characterized. Originally, virtual interrupt makes a guest OS fully preemptible because guest OS cannot disable physical interrupts, and allows preemption at arbitrary points. Namely, all the guest OSs run in user-level, and interrupt is disabled only within the minimal code inside the hypervisor. Thus, the entire system becomes responsive because a guest OS can be preempted even though it disables virtual interrupts. Furthermore, misbehavior with handling interrupt effects locally, and never affects the other domains.

However, virtualized interrupt handling can cause additional delay since a guest OS cannot fully control physical interrupts. The driver domain can be preempted by a user domain even though it has pending interrupts. Note that the hypervisor can still receive physical interrupts although a guest OS disables virtual interrupts. If the guest OS disables virtual interrupts, the received virtual interrupts are stored in the event channel, and the guest OS handles pending interrupts only after it re-enables virtual interrupts again. In this situation (that the driver domain has pending virtual interrupts while disabling virtual interrupts), if the hypervisor receives additional physical interrupts that might trigger inter-VM scheduling, the driver domain would be scheduled out. Consequently, the pending virtual interrupt handling is delayed until the driver domain is re-scheduled by the hypervisor, and the driver domain re-enables virtual interrupts.

For example, in an idle paravirtualization of XenoLinux, a guest OS disables interrupts just before the entering the hypercall, *do_block()*, as shown in Figure2, and re-enable after the call in order to reduce unnecessary wake ups during the idle.

If an interrupt occurs just after it is disabled, then the interrupt handling is delayed until the guest OS re-enables interrupts. Since *arch_idle()* invokes a hypercall that blocks itself, the domain is scheduled out. The domain has to wait until it receives additional event at a later time. Namely, the pending interrupt is handled with a considerable delay, which will show as latency spikes in experiment results.

| vcpu state | priority | intr. state | sched. out count |
|---|---|---|---|
| Blocked | BOOST | Enabled | 0 |
| | | Disabled | 275 |
| | UNDER | Enabled | 0 |
| | | Disabled | 13 |
| Unblocked | BOOST | Enabled | 0 |
| | | Disabled | 664,190 |
| | UNDER | Enabled | 8 |
| | | Disabled | 41 |

**Table 1: Schedule out counts for network interrupts**

This contradicts to the original intention of interrupt virtualization because the interrupt handling can be delayed by virtualization even though virtual interrupt handling makes the guest OS more preemptive. Since no physical interrupt are blocked by a guest OS (although virtual interrupts are disabled within the guest OS), the guest OS has become fully preemptive. However, it does not lead to the reduced interrupt handling latency because the physical interrupts are still received by the hypervisor; and some interrupts allows another guest domain to preempt the driver domain even though it has pending interrupts to process.

### 3.1.3 Experiments

To analyze the delayed interrupt handling at the driver domain, we measure the actual latency between physical interrupt handler (i.e. *handle_interrupts()* ) at Xen and the netback handler (i.e. *net_be_start_xmit()* ) at the driver domain. The experimental parameters are the same as in Section 3. To categorize the reasons of delay spikes, we count the number of different cases when the driver domain is schedule out to another domain. For each schedule out events, we check the VCPU states 1) whether it is blocked or not, 2) the priority is BOOST or UNDER, and 3) whether virtual interrupts are disabled or not. If the VCPU is blocked, and interrupt is disabled, then it implies that the driver domain invoked idle with virtual interrupts disabled. This causes the delay by virtualized interrupt handling. If the VCPU is unblocked and BOOST, it implies multi-boost because there should be another VCPU that has BOOST priority. If VCPU is unblocked and UNDER, it implies that the VCPU was not boosted because it was on the run queue.

Table 1 summarizes the results for 343,335 ping tests. Among total 1,003,321 number of interrupts, 6,327 interrupts are handled after 1ms, which is regarded as delay spike. That is, about 0.63 percent of network interrupts experiences large interrupt handling latency more than 1ms. Table shows that the virtualized interrupt handling causes 288 (= 275+13) number of schedule outs, multi-boost causes more than 0.6 million schedule outs, and not-boosting VCPUs causes 49 (= 8 + 41) schedule outs.

## 3.2 Netback-domU Latency

Besides Xen-netback latency, another large latency can occur between netback driver and frontend driver. The major reason for this latency is due to *non-boosted VCPU* of the credit scheduler. The credit scheduler doesn't boost a

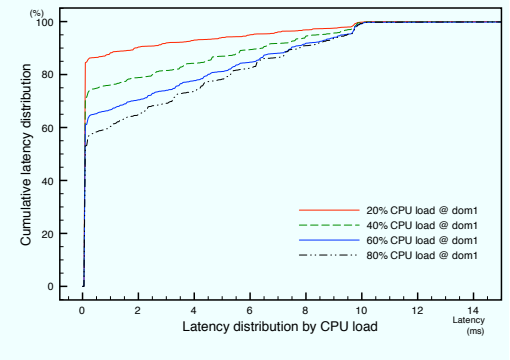


**Figure 3: Impact of CPU load on I/O latency**

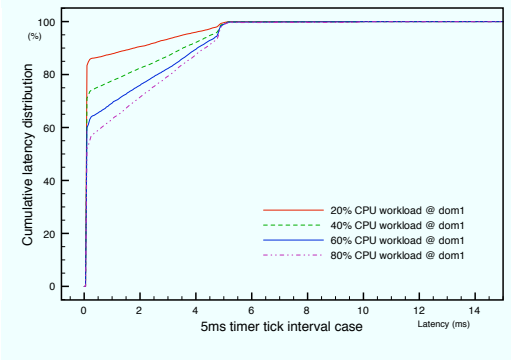


**Figure 4: Timer tick interval and I/O latency**

VCPU when the designated VCPU is in the run queue or has consumed all its credit. Xen-ARM regards those VCPUs as CPU-bounded, and doesn't boost them so that the CPU fairness can be preserved. In reality, a guest OS usually has CPU-I/O mixed workloads, and long I/O latency is observed when CPU-intensive job is running. Thus, I/O latency increases as the CPU utilization of a user domain. Namely, when a user domain utilizes more CPU, then its I/O latency is very likely to be longer. The following experiment shows the impact of CPU utilization to the I/O latency of a domain.

We run three guest OSs as follows: Dom0 for the driver domain, dom1 for CPU-I/O mixed workload, and dom2 for CPU-intensive workload to ensure the work-conserving. Then, we measured I/O latency from the interrupt handler within Xen-ARM to the netfront driver at dom1 by varying the CPU utilization (20%, 40%, 60%, 80% of CPU load) at dom1. The graph in Figure3 shows the distribution of the measured latency.

According to the graph in Figure3, latency could be increased by CPU utilization of dom1. In the case that dom1 has 20% CPU utilization, 85% of interrupts are handled immediately. The rest 15% of interrupts are delayed because the VCPU has not been boosted by CPU workload. On the other hand, for 80% CPU utilization case, only 50% of interrupts are handled immediately, which means that the probability to have small latency decreases according to the CPU utilization.

Delay from CPU workload is closely related with timer tick interval. In general, system becomes less responsive and the overall performance overhead increases as timer tick interval increases. Namely, if the timer tick interval is small, scheduler is more frequently invoked, and scheduler is able to dispatch the most urgent thread as soon as possible. However, frequent context switch introduces additional overhead from cache pollution, TLB flush, etc. The same rule applies to inter-VM scheduling in Xen-ARM. As we can see in the Figure 3, the latency is bounded by timer tick interval, 10ms.

Figure 4 shows the same experiments with 5ms timer tick interval. As we can see in the figure, the delay impact from CPU-bound work also decreases as the timer tick interval.

# 4. I/O SCHEDULING FOR REAL-TIME GUEST OS

As we have observed in the previous section, the current credit scheduler is not suitable for supporting real-time I/O in its current form. Two major latencies are 1) Xen-netback latency, and 2) netback-domU latency. To bound the interrupt handling latency due to Xen-netback latency at Xen-ARM, we introduce the following methods to the hypervisor so that the designated backend driver can handle interrupts in a timely manner.

Firstly, we mitigate the latency by virtualized interrupt handling. We changed the Xen-ARM's scheduler so that the driver domain is not scheduled out when the driver domain disables virtual interrupts. Since the driver domain is not scheduled out during it disables interrupts, it can re-enable interrupts as soon as possible, and backend driver can immediately process the pending interrupts. This resolves all the virtual interrupt-related problems including wrong idle paravirtualization. Note that we do not disable physical interrupts, but pinning the driver domain when it disables virtual interrupts.

Secondly, we differentiate the interrupt handling for real-time I/O devices (i.e. netdevice in our case) from the other devices. We use FIQ[2] of ARM processor so that the real-time devices take advantage of architectural support. FIQ has higher priority than IRQ, and this largely removes misbehavior through disabling interrupts. For example, the driver domain is able to handle network interrupt with FIQ, even in code with IRQ disabled.

Thirdly, we further optimized the Xen-ARM's scheduler so that the driver domain can be scheduled at the highest priority particularly when it has pending interrupts. The highest priority is remained until the domain finishes the interrupt handling, and the inter-VM scheduling is postponed until the driver domain triggers end-of-interrupt via a hypercall *pirq_guest_eoi()*.

Figure 5 shows the enhanced I/O latency by our modifications. Three domains run at the same time: dom0 for driver domain, dom1 for handling icmp requests, and dom2 for cpu-intensive operation for ensuring work-conserving scheduling policy. Each line presents the cumulative distribution for the presented delay spikes. (1), (2), and (3) represent our modifications regarding virtual interrupt preemption, FIQ handling, and prioritized driver domain scheduling, respectively. Originally, 0.8% interrupts are handled over 1ms, (2) and (3) reduce the spikes to 0.71% and 0.65%, respectively.

Finally, we modified the Xen-ARM's scheduler so that it can schedule domains with strict priority, based on urgency. By providing the strict scheduling priorities, the interrupt la-

[2]FIQ is an ARM-specific architectural mode for faster interrupt handling

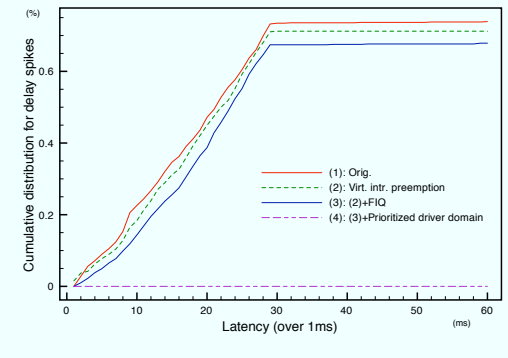


**Figure 5: Reduced delay spikes within Xen-netback**

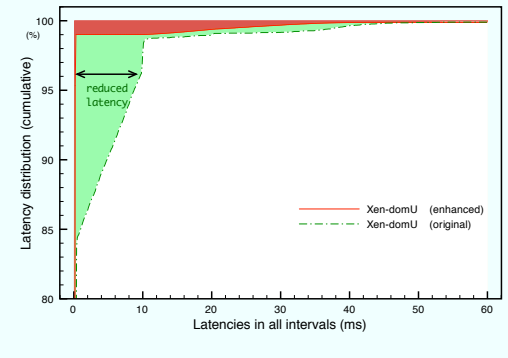


**Figure 6: Latency enhancement by proposed methods**

tency can be deterministically bounded. Priority orders are: 1) driver domain within interrupt, 2) dom1 within interrupt context, 3) dom1 without interrupt context, 4) driver domain without interrupt context, and rest of the cases are handled based on the existing credit scheduler's priority. Strict priority-based scheduling overcomes the two major limitations in the current credit scheduler (from multi-boost and non-boosted VCPU). It can distinguish the urgency among the BOOST domains, and eliminates I/O latency affected by CPU load. Therefore, a real-time guest OS can handle interrupts with strict time-bound while retaining the split driver model.

The overall cumulative latency distribution is presented in Figure 6. In the graph, more than 99% of network interrupts are handled within 1ms at dom1. Note that dom1 has 20% CPU load, and rest 1% interrupts are measurement error because of the complex interrupt generation of the device. Considering this measurement error, our modification guarantees interrupt handling with 1ms time-bound. Finally, after applying all the modifications, all the delay spikes are eliminated for one million of network interrupts.

# 5. CONCLUSION AND FUTURE WORK

This paper addresses I/O latency in a Xen-ARM-based virtual machine. To support a hard real-time guest OS, the hypervisor has to ensure that a real-time guest OS can handle I/O with strict time-bound. The current Xen-ARM has limitations regarding its scheduler and unique split driver model. BOOST priority in credit is easy to be negated, and domains are sometimes not boosted. In addition, virtual interrupt handling at the driver domain can be delayed because another domain can preempt the driver domain even within the ISR. To guarantee time-bounded latency, we modified the hypervisor scheduler in order to give strict priorities to domains. With the modified schedulers, we achieved guaranteed I/O handling within 1ms time-bound. The result shows that the real-time I/O handling can be possible with Xen-ARM.

Our prototype implementation focuses on single-core platform, but we believe that our mechanisms are orthogonally applicable to multicore systems. Fairness is another concern in heavy I/O environment, and we are considering fairness among multiple VMs with different virtual I/O devices environment as future work.

# 6. ACKNOWLEDGEMENT

The authors appreciate reviewers for their helpful comments and feedback. This research work was financially supported by the Seoul Research and Business Development Program (No. WR080951) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0029180).

# 7. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, pages 164–177. ACM, 2003.

[2] G. Heiser and B. Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[3] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110, New York, NY, USA, 2009. ACM.

[4] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 97–108, New York, NY, USA, 2010. ACM.

[5] S.-M. Lee, S.-B. Suh, B. Jeong, S. Mo, B. M. Jung, J.-H. Yoo, J.-M. Ryu, and D.-H. Lee. Fine-grained i/o access control of the mobile devices based on the xen architecture. In *MobiCom '09: Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 273–284, New York, NY, USA, 2009. ACM.

[6] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[7] S. Sumpf and J. Brakensiek. Device driver isolation within virtualized embedded platforms. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5, Jan. 2009.

# Keynote - Peter Druschel (MPI-SWS)

He is the founding director of the Max Planck Institute for Software Systems. He also leads the distributed systems research group. Prior to joining the MPI-SWS in August 2005, Peter was a Professor of Computer Science at Rice University in Houston, TX. He also spent time with the SRC group at Laboratoire d'Informatique de Paris 6 (LIP6) (May-June 2000, June 2002), the Cambridge Distributed Systems group at Microsoft Research Cambridge, UK (August-December 2000), and the PDOS group at the MIT Laboratory for Computer Science (January-June 2001). He is the recipient of an NSF CAREER Award (1995), an Alfred P. Sloan Fellowship (2000), and the 2008 Mark Weiser Award. He is on the editorial boards of the Communications of the ACM (CACM) and the ACM Transactions on Computer Systems (TOCS) and has served as program chair for SOSP, OSDI and NSDI. Together with Antony Rowstron and Frans Kaashoek, he started the IPTPS series of workshops. Peter is a member of the Academia Europaea and the German Academy of Sciences Leopoldina. Peter received his Ph.D. from the University of Arizona in 1994, under the direction of Larry L. Peterson.

# One Optimized I/O Configuration per HPC Application: Leveraging the Configurability of Cloud

Mingliang Liu, Jidong Zhai and Yan Zhai
Department of Computer Science and Technology, Tsinghua University
{liuml07,zhaijidong,zhaiyan920}@gmail.com

Xiaosong Ma
Department of Computer Science, North Carolina State University
Computer Science and Mathematics Department, Oak Ridge National Laboratory
ma@csc.ncsu.edu

Wenguang Chen
Department of Computer Science and Technology, Tsinghua University
cwg@tsinghua.edu.cn

## ABSTRACT

There is a trend to migrate HPC (High Performance Computing) applications to cloud platforms, such as the Amazon EC2 Cluster Compute Instances (CCIs). While existing research has mainly focused on the performance impact of virtualized environments and interconnect technologies on parallel programs, we suggest that the configurability enabled by clouds is another important dimension to explore.

Unlike on traditional HPC platforms, on a cloud-resident virtual cluster it is easy to change the I/O configurations, such as the choice of file systems, the number of I/O nodes, and the types of virtual disks, to fit the I/O requirements of different applications. In this paper, we discuss how cloud platforms can be employed to form customized and balanced I/O subsystems for **individual** I/O-intensive MPI applications. Through our preliminary evaluation, we demonstrate that different applications will benefit from individually tailored I/O system configurations. For a given I/O-intensive application, different I/O settings may lead to significant overall application performance or cost difference (up to 2.5-fold). Our exploration indicates that customized system configuration for HPC applications in the cloud is important and non-trivial.

## 1. INTRODUCTION

Cloud computing is gaining popularity in many areas, including High Performance Computing (HPC). Beside being CPU-intensive, HPC applications are sensitive to network bandwidth/latency as well as the performance of I/O storage systems, making them particularly challenging to run efficiently in clouds. Amazon EC2, the leading IaaS provider, released *cluster compute instances (CCIs)* in July 2010 [3,6], to explicitly target HPC applications by offering dedicated physical node allocation, powerful CPUs, and improved interconnection.

Even with such new cloud platforms provided for HPC workloads, the efficiency and cost-effectiveness of cloud computing for HPC applications are still to be determined [7]. For example, our recent evaluation indicates that although the EC2 CCIs are able to provide significantly better performance compared to earlier instance classes, small message dominated applications suffer from the long communication latency on cloud platforms [1]. Apparently, the feasibility of cloud computing is highly workload-dependent and heavily depends on individual applications' characteristics. Therefore, an important question is: ***What kind of applications are suitable to execute in cloud rather than using traditional resources (i.e., supercomputers or in-house small clusters)?***

In assessing HPC applications' suitability for cloud execution, most existing studies have focused on the the performance impact of virtualized environments and interconnect technologies on cloud platforms. In this paper, we suggest that the unique configurability provided by cloud is another important dimension to consider. To make our discussion succinct and clear, in this paper we limit our discussion to I/O configurability on I/O-intensive HPC applications, though the topic can be extended to other application domains and beyond I/O.

Besides the well-known elasticity in resource acquisition, usage, and releasing, cloud platforms provide a level of configurability impossible with traditional HPC platforms. Unlike supercomputers or in-house clusters, which only offer fixed, one-size-fits-all configurations, clouds allow users to customize their own virtual cluster and perform reconfiguration at the user, application, or even job level. For example, on traditional parallel computers the users will not be able to choose or configure the shared or parallel file system, but on EC2 one can easily set up a personalized I/O subsystem. For example, users can install and select from multiple file systems such as NFS, PVFS [4], and Lustre [5]. For a selected parallel file system, they can configure the number of I/O servers and their placement. Further, users can


*APSys* '2011 Shanghai, China

choose from different types of low-level storage devices, the ephemeral disk and Elastic Block Store (EBS), or use a combination of them. Changes in configuration can typically be deployed quickly, with simple modifications made to scripts. Therefore, the cloud enables users to setup optimized I/O configurations for **each** application upon its execution, instead of configuring an I/O sub-system for **all** applications that may run in the system. Intuitively, the better configurability of cloud should give it competitive advantage against in-house clusters and supercomputers.

To demonstrate this, we perform a few experiments and conduct related analysis on Amazon EC2 CCIs to see how different I/O sub-system configurations could affect the performance of I/O-intensive MPI applications. We also use several micro-benchmarks to evaluate the cloud storage devices and different file systems, which help us understand the low level I/O behavior in cloud systems and prune the experimental space.

This work distinguishes itself from past investigations on HPC in the cloud in the following aspects:

- We identify the configurability enabled by cloud platforms as an important factor to consider, especially for I/O-intensive HPC applications.
- We conduct a qualitative discussion on the storage system configuration options on cloud platforms and their potential impact on HPC application performance as well as user experience.
- We evaluate two applications with different I/O configurations and observe large performance gaps between these configurations. The result suggests that per-application optimization on cloud I/O configuration deserves further investigation.

The rest of the paper is organized as follows. Section 2 introduces the Amazon EC2 CCI storage system and evaluation of multiple storage options on it. Section 3 shows our preliminary result of two applications with different I/O sub-system configurations. Section 4 discusses related work and conclusion is given in Section 5.

# 2. STORAGE SYSTEM CUSTOMIZING OPTIONS IN THE CLOUD

In this section, we discuss several key parameters and related issues in setting up customized storage sub-system on cloud-based virtual clusters.

## 2.1 File System Selection

A parallel or shared file system is indispensable for parallel program execution, which provides a unified persistent storage space to facilitate application launching, input and output file accesses, and parallel I/O support. Typically, supercomputers or large clusters are equipped with parallel file systems such as Lustre [5], GPFS [11], and PVFS [4], while smaller clusters tend to choose shared/distributed file systems such as NFS. However, end users seldom have the option of choosing or configuring file systems on traditional HPC facilities. One outstanding advantage of cloud HPC execution is that users can choose a parallel or shared file system based on individual applications' demands, and can switch between different selections quite easily and quickly.

In choosing an appropriate file system, users need to consider their applications' I/O intensity and access pattern. For example, a simple NFS installation may suffice if the application has little I/O demand, or a low I/O concurrency (seen in parallel codes where one process aggregates and writes all output data, a rather common behavior in simulations). In contrast, applications that write large, shared files usually benefit from parallel file systems, especially those heavily optimized for MPI-IO operations. Especially, parallel file systems will allow users to scale up the aggregate I/O throughput by adding more I/O servers, while the single NFS server may easily become a bottleneck under heavy I/O loads. Depending on the variability among per-file access patterns, users may elect to use file systems that give extra flexibility in performance optimization, such as the per-file striping setting allowed by PVFS.

In our preliminary evaluation presented in Section 3, we demonstrate the performance difference of an I/O intensive parallel program running on NFS and PVFS, two popular file systems on clusters. Due to the space limit, we confine our discussion here to these two file systems. However, we believe that the potential to gain performance and/or cost benefits via storage option tuning apply to other file system choices as well.

## 2.2 Storage Device Selection

Another important storage parameter is the type of underlying storage devices. Cloud platforms typically provide multiple storage choices, with different levels of abstraction and access interfaces. For example, with EC2 CCIs, each instance can access three forms of storage: (1) the local block storage ("ephemeral") with 1690GB capacity, where user data are not saved once the instances are released, (2) off-instance Elastic Block Store (EBS), where volumes can be attached to an EC2 instance as block storage devices, whose content persist across computation sessions, and (3) Simple Storage Service (S3), Amazon's key-value based object storage, accessed through a web services interface.

For HPC applications, the ephemeral and EBS are more apparent choices as storage devices, as they do not require modifications to applications. S3, on the other hand, is designed more for Internet or database applications and lacks general file system interfaces needed in HPC programs.

Beside data consistency, the ephemeral and EBS devices possess different performance characteristics and usage constraints. One instance can only mount up to two ephemeral disks, while the number of EBS disks attached to it can be almost unlimited. In our preliminary benchmarking, we found the performance gap between the EBS and ephemeral disks small, especially for writes (the major I/O operation in numerical parallel programs), as to be demonstrated in Figure 1. However, we observed that the ephemeral disk has better availability and lower performance variance. This may be the result of different disk affinity and different virtualization techniques. Finally, the ephemeral disks are free, while EBS disks are charged by the storage capacity consumed.

Therefore, the choice between these two storage devices again depends on the needs of individual applications or users. For instance, production runs that generate results to be migrated to and visualized on users' local platforms may get the best benefit from using ephemeral disks, as persistent storage is not needed. On the other hand, repeated

processing of the same dataset (such as sequence database searches) will benefit from using the EBS. As another example, bandwidth-thirsty applications relying on parallel I/O with file striping may suffer from the load imbalance introduced by the high performance variability of EBS disks.

## 2.3 File System Internal Parameters

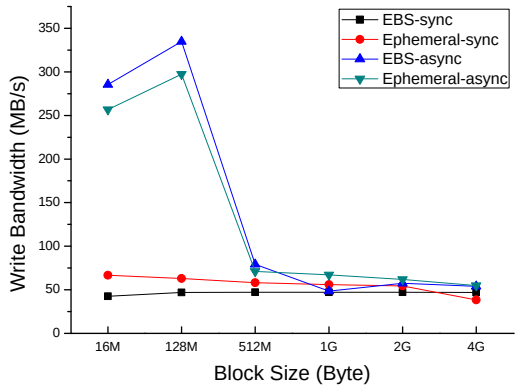


**Figure 1: NFS write bandwidth, measured with IOR**

For NFS, one sample internal parameter is the choice of write modes: *sync* vs. *async*. Figure 1 shows the results from running IOR [12], a parallel I/O benchmark, with these two modes. We measured the NFS write bandwidth, using two client instances, each running 8 IOR processes. The 16 processes write a shared file collectively, each dumping a contiguous partition (block). The block size is varied from 16MB to 4GB. As shown in Figure 1, with small data blocks, the *async* mode is able to offer a significant performance advantage, due to the buffering effect at server side. It will be a sound choice for applications that output a moderate amount of data and do not require that such data are flushed to secondary storage, such as periodic checkpoints (whose loss can be compensated by re-executing a certain number of computation timesteps in most numerical simulations). We use the *async* mode for NFS server by default in the following experiments.

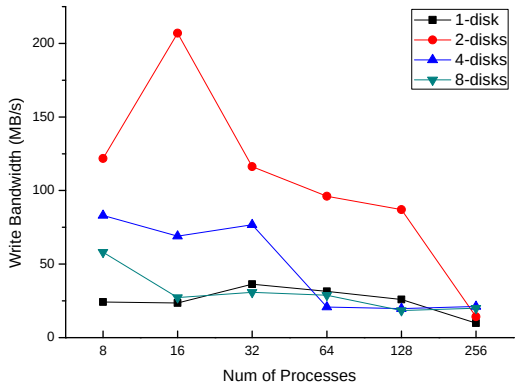


**Figure 2: NFS write bandwidth with EBS disks combining into a software RAID0**

On cloud platforms, a user can have tremendous flexibility in scaling up the aggregate I/O capacity and bandwidth, according to the I/O demands of their applications. For parallel file systems, this can be done by increasing the number of I/O servers, while the NFS could not scale because of the single server bottleneck. For all file systems, we can also improve each I/O server's capability by increasing the number of disks attached to it. One simple way of doing this is to combine multiple disks into a software RAID0 partition. However, one may not obtain expected scaling effect when adding (and paying for) more disks to an I/O server. Figure 2 illustrates this with another set of IOR collective write experiments, where we fixed the size of data block per process (2GB) and varied the number of processes. We evaluated different numbers (1, 2, 4, 8) of EBS disks forming a single RAID0 partition on the NFS server, and found that the system hits its peak bandwidth when only two disks are mounted. There are many possible reasons for this, such as the virtualized layer between storage volumes and multiple users, the network connection between the physical node and the physical storage devices, and the performance variance of EBS devices. In our future work we plan to investigate the disk scaling problem, as a part of automatic cloud storage configuration.

In addition, each file system has many parameters (such as striping factor, unit size, metadata server placement and disk synchronization), most of which can be configured the same way on both the cloud and the in-house cluster platforms. However, there are exceptions due to the difference in usage model brought by the virtualized cluster environment on clouds. For example, the compute resource of an I/O server is not fully utilized if the I/O server process occupies an 8-core virtual machine. Unlike on a static cluster, where an I/O server is possibly shared among multiple applications and/or users, a cloud-based virtual cluster is typically used for a dedicated run. Therefore, it is an interesting problem to explore the I/O server and metadata server placement on cloud instances, to achieve better resource utilization and overall cost-effectiveness. In Section 3 we will report empirical results with sample placement settings.

# 3. PRELIMINARY APPLICATIONS RESULTS

## 3.1 Platform Description

Our tests use the new HPC-oriented Amazon EC2 Cluster Compute Instances (CCIs) available since July 2010. Each CCI has 2 quad-core Intel Xeon X5570 "Nehalem" processors, with 23GB of memory. Each instance is allocated to users in a dedicated manner, unlike those in most other EC2 instance classes [3]. The CCIs are interconnected with 10 Gigabit Ethernet. Due to the unstable availability of EBS storage that we recently encountered, all our experiments in this section use the ephemeral disks.

Regarding software settings, we use the Cluster Instances Amazon Linux AMI 2011.02.1 associated with CCIs, the Intel compiler 11.1.072 and Intel MPI 4.0.1. The default compiler optimization level is $-O3$.

## 3.2 Selected HPC Applications

Our experiments evaluated two parallel applications. BTIO is the I/O-enabled version of the BT benchmark in NPB suite [13]. The program solves Navier-Stokes equations in three spatial dimensions, which are discretized, unsteady and compressible. Each of the processor will be in charge of multiple Cartesian subsets of entire data set. The I/O

strategy employs collective buffering and writing via MPI-IO interface, to avoid heavy fragmentation caused by writing per-process output files. The default I/O frequency is used, appending to the shared output file every 5 computation time steps, resulting in an output file sized around 6.4GB. The BT problem size was class C for all the tests, built with `FULL` subtype. POP is an ocean circulation model which solves the three-dimension primitive equations for fluid motions on the sphere [9] and exhibits a read-once-write-many I/O pattern. It reads a setup file at the initialization phase and performs write operation in native Fortran I/O interface at each period time step. We used the POP grid dimensions of 320x384x20 and the total output size is about 6GB, aggregated and written by process 0.

## 3.3 BTIO Results

In this section, we present the performance and total computation cost of running BTIO on EC2 CCIs, using NFS (*async* mode) and PVFS respectively. For each storage option, we also present the performance difference between two I/O server placement modes: I/O servers occupy separate compute instances under the *dedicated* mode, and co-reside instances with compute processes under the *part-time* mode.

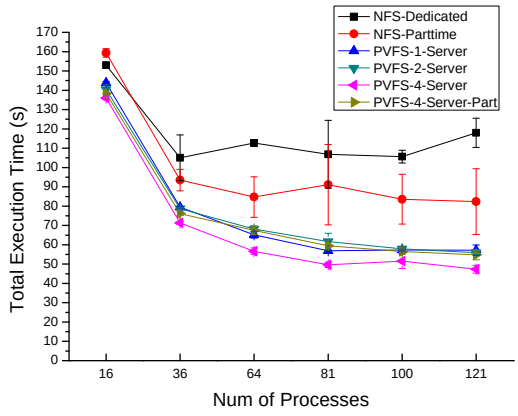


**Figure 3: BTIO total execution time**

Figure 3 shows the total execution time of BTIO under different file system configurations. There are up to 4 I/O servers, each mounting two ephemeral disks with software RAID0. For PVFS, one of the I/O servers acts as the metadata server. From the results, we can see that for BTIO, PVFS beats NFS in all test cases. For example, there is an up to 60% performance improvement from using NFS (with one dedicated server) to using PVFS (with 4 dedicated servers). The performance gain is more likely due to the collective MPI-IO optimizations contained in the PVFS design, as even with just one I/O servers, PVFS significantly outperforms NFS. Also, the NFS performance appears to have a much higher variance compared to that of PVFS. Interestingly, NFS performs better when the I/O server runs in the part-time mode, which can be partly attributed to the enhanced data locality and reduced network contention. However, we doubt that this factor alone causes the large performance difference observed and are carrying out more detailed investigation.

Judging by the I/O bandwidth measured from BTIO, PVFS performance scales well with increased number of I/O servers (results not plotted due to space limit). However, the scalability does not reflect well in Figure 3, the overall performance chart. This is due to the total portion of execution time devoted to I/O. For this strong-scaling experiment we expect to see that I/O gains more weight in the execution time, but the computation component of BT does not scale well on Amazon EC2, due to the unsatisfactory interconnection. Therefore as the number of processes increases, the total percentage of time spent on I/O is maintained at around 10%.

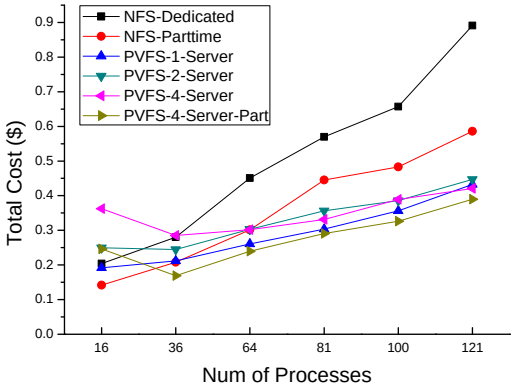


**Figure 4: The total cost of BTIO executions**

A related issue is the total cost of execution, when we factor in the total number of instances needed. Figure 4 shows the cost comparison derived from the BTIO results, using the Amazon charging rate of $1.6 per instance per hour. Here we can see the appeal of using part-time I/O servers. In addition, at a small execution scale, NFS with one part-time server actually appears to be the most cost-effective option.

## 3.4 POP Results

We run POP with similar I/O configurations (Figure 5). Unlike with BTIO, here NFS (*async* mode at server side) outperforms PVFS across all configurations. There are several possible reasons. First, POP has process 0 carry out all I/O tasks, making the parallel I/O support that PVFS was designed for less useful and the network communication more bottleneck-prone in I/O. Second, POP does I/O via POSIX interfaces, which is not optimized in PVFS. Due to its heavy communication with small messages, POP does not scale on EC2, as can be seen from Figure 5. However, its I/O behavior is still representative in parallel applications.

# 4. RELATED WORK

Recently, there have been a series of research efforts focusing on the performance of using public clouds for high performance computing. Most of previous work are focusing on computation and communication behavior running on clouds. Very little work has investigated I/O and storage issues in virtualized or cloud environments. Some of them focused on long-term storage or inter-job or inter-task data flow. For example, Juve et al. studied the performance and cost of different storage options for scientific workflows running on Amazon EC2 [8]. Palankar et al. assessed the feasibility of using Amazon S3 for scientific grid computing [10]. Abe et al. constructed pWalrus, which provides

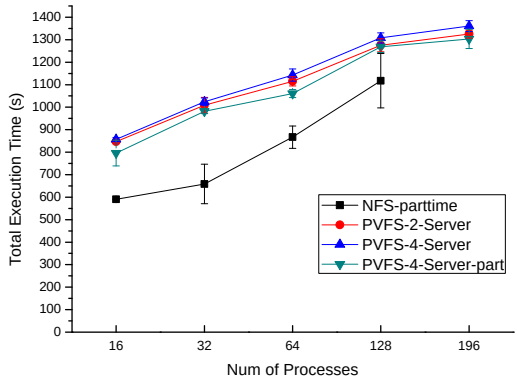


**Figure 5: POP total execution time**

S3-like storage access on top of the PVFS parallel file system on an open-source cloud [2]. Yu and Vetter studied parallel I/O in Xen-based HPC, but the environment used only have virtualized compute nodes and the authors only tested at most 16 processes [14]. To our best knowledge, our work is the first to evaluate different parallel/shared file system settings for parallel I/O on public clouds.

# 5. CONCLUSION

In this paper, we demonstrate that the unique configurability advantage offered by public clouds may bring opportunities for HPC applications to achieve significant performance or cost improvement. In particular, we illustrate the impact of virtual cluster configurability by assessing the impact of I/O system customization on the Amazon EC2 system. Our results indicate that for I/O-intensive applications, cloud-based clusters enable users to build per-application parallel file systems, where no one-size-fits-all parallel I/O solution can satisfy the needs of all applications. In the future, we will try to explore how the storage options interact with the characteristics of HPC applications and finally to automate the process of choosing I/O storage options for an individual application.

# 6. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable feedback. We'd like to thank Scott Mcmillan, Bob Kuhn and Nan Qiao from Intel for their interest, insights and support. The research was supported by National Natural Science Foundation of China under Grant No. 61073175. It was also supported in part by NSF grants 0546301 (CAREER), 0915861, 0937908, and 0958311, in addition to a joint faculty appointment between Oak Ridge National Laboratory and NC State University, as well as a senior visiting scholarship at Tsinghua University.

# 7. REFERENCES

[1] Technical report r2011.4.10. Technical report, Tsinghua University. `http://www.hpctest.org.cn/resources/cloud.pdf`.

[2] Y. Abe and G. Gibson. pWalrus: Towards Better Integration of Parallel File Systems into Cloud Storage. In *Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2010.

[3] Amazon Inc. High Performance Computing (HPC). http://aws.amazon.com/ec2/hpc-applications/, 2011.

[4] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4*, pages 28–28. USENIX Association, 2000.

[5] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. http://www.lustre.org/docs/whitepaper.pdf, 2002.

[6] N. Hemsoth. Amazon adds hpc capability to ec2. HPC in the Cloud, July 2010.

[7] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 159–168. IEEE, 2010.

[8] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Data sharing options for scientific workflows on amazon ec2. In *SC'10*, pages 1–9, 2010.

[9] LANL. Parallel ocean program (pop). `http://climate.lanl.gov/Models/POP`, April 2011.

[10] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for Science Grids: A Viable Solution? In *Proceedings of the International Workshop on Data-Aware Distributed Computing*. ACM, 2008.

[11] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, 2002.

[12] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC'08*, page 42. IEEE Press, 2008.

[13] P. Wong and R. der Wijngaart. Nas parallel benchmarks i/o version 2.4. *NASA Ames Research Center Tech. Rep. NAS-03-002*, 2003.

[14] W. Yu and J. S. Vetter. Xen-Based HPC: A Parallel I/O Perspective. In *IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2008.

# SLIM: Mmap from the Cloud to Device, and Back

Jinghao Shi[◦] Mingyuan Xia[†] Ming Wu[‡] Lintao Zhang[‡] Zheng Zhang[‡]
[◦] University of Science and Technology of China [†] McGill University, Canada [‡] Microsoft Research Asia
jhshi89@gmail.com mingyuan.xia@mail.mcgill.ca
{miw,lintaoz,zzhang}@microsoft.com

## ABSTRACT

In the era of cloud computing, mobile applications often need to leverage a new storage hierarchy that includes not only the traditional main memory and secondary storage on devices, but also storage and computation capabilities from the cloud. In this paper we propose a new data structure library called SLIM. SLIM provides familiar STL-like interfaces and abstractions while accommodates the storage hierarchy that transcends device/cloud boundary. Initial evaluation shows that using SLIM can greatly simplify application development and reduce network and energy cost compared with traditional approaches.

## Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Client/server*; C.4 [**Performance of Systems**]: Design studies; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

## General Terms

Design, Performance

## Keywords

Mobile, cloud, data structure, STL, storage hierarchy

## 1. INTRODUCTION

More computing devices are becoming mobile. These devices derive much of their values by being connected not only to each other but also to the cloud. Leveraging the cloud for mobile devices is becoming an important research topic. Broadly speaking, cloud is seen not only as a data source and sink, but increasingly as computation backbone as well.

Splitting computation across device and cloud boundary is complex and challenging. It touches on many known hard problems in distributed computing, including state maintenance, heterogeneous computing environment and programming model. The common theme in much of the previous work is to preserve legacy compatibility as much as possible.


*APSys 2011,* July 11–12, 2011, Shanghai, China


However, interesting mobile applications often starts anew on the mobile devices, whereas time-tested important desktop applications (e.g. email and browser) are worth the effort to be rewritten. Therefore, we do not believe that preserving backward compatibility is a pressing issue. The SLIM project is motivated by the observation that programming abstraction evolves much slower than the hardware trend. We argue that deriving program logic from data structures is a "constant" despite hardware and software changes. Indeed, modulo various marshaling and serialization steps in between, mobile applications all start their life as data structures manipulated by logic in the cloud and end up as data structures on the devices.

We attempt to re-examine the notion of data structures in light of the new storage hierarchy. Mosting existing data structure libraries like STL (Standard Template Library [12]) considers only a single layer: the volatile main memory. In the mobile world, there are actually two more layers of storage: flash and the cloud. The characteristics of these three layers are markedly different. The main Memory is fast but volatile, flash is large and persistent, while cloud has infinite computing and storage capacity (with respect to the device), but connectivity to it can be intermittent and unreliable. The goal of the SLIM project is to investigate the feasibility of building a high performing set of data structures that transcend these boundaries while preserving the semantics of existing libraries (e.g. STL) as much as possible.

Our contributions include:

- We propose an STL-like data structure abstraction for mobile/cloud application development. While preserving most of the STL abstractions and interfaces, SLIM gives programmers simple directives to explicity control the data structures such as the storage layers involved, the data flow direction of updates, and the consistency bounds.
- We propose several techniques to make SLIM efficient and easy to use. SLIM employs an efficient pointer swizzling algorithm to handle the pointer-rich nature of data structures across memory hierarchy boundaries. To deal with the unreliable nature of communication between cloud and device, we leverage advanced asynchronous programming model called "promoise" to simplify failure handling for the programmers.
- By building an RSS reader as an example, we demonstrate that SLIM data structures provide a performance benefit, and have the potential to enable computation offloading to the cloud.

The rest of the paper is organized as follows. Section 2 describes the programming model. Section 3 gives a high-level overview of the system. Section 4 presents preliminary results. Section 5 and 6 covers related work, discussion and future work.

## 2. THE PROGRAMMING MODEL

The goal of the SLIM family of data structures is to create as close as possible the illusion of dealing with the traditional and local STL data structures, whereas the runtime hides the interactions with the memory hierarchy. To begin the discussion, we will start with a piece of pseudo-code for a RSS reader using the STL library, as follows:

```
//Step1: fetch RSS xml file to local
XMLObject xmlObj = XML::Load("RSS URL");
//Step2. parse XML to construct feed array
std::vector<FeedItem> feeds = parse(xmlObj);
//Step3. find and display new feeds
foreach (feed in feeds) {
    if (/*feed is new*/)
        display(feed);
}
Sleep(freshInterval);
```

The code snippet above retrieves an XML object that contains the content of the current feeds, and then parse it into an STL vector for display. The reader is stateless, as such redundant feeds must be removed. The corresponding SLIM version is shown below:

```
//SLIM vector of all feeds, parsed and to be updated by the cloud
SLIMVector<FeedItem, L3, C2D, OnDirtyBound<1> > feeds("RSS URL
    ");
//set callback function for new feeds (use lambda function for simplicity)
feeds.onUpdate([] (FeedItem newFeeds[]) {
    //all new feeds, just display
    foreach (feed in newFeeds)
        display(feed);
});
```

A SLIM data structure takes three directives at the declaration time to tell the runtime the desired behavior of interacting with multiple storage layers: `Level`, `Direction` and `Consistency`:

- `Level` can be L1, L2 and L3. An L1 vector is exactly the same as its volatile in-memory STL counterpart. An L2 vector corresponds to the external-memory model [13] where the local persistent storage (usually Flash) is included as the bottom layer. L3 indicates that the cloud is the final layer, which is the case for the SLIM-based RSS reader. In our current design, the layers are inclusive. In other words, an L3 SLIM vector also can store data in local Flash storage if it cannot all fit in memory. We do not see immediate need to exclude local secondary storage explicitly in the L3 case, though implementing such a policy is trivial.

- `Direction` expresses the data flow direction for updates. `D2C` is for cases where the updates are coming from devices towards the cloud, and is useful for scenarios such as the outgoing queue for an Instant Messenger client, or a vector that stores data obtained by a sensor node for cloud to analyze. `C2D` is the opposite of D2C, where updates are first collected in the cloud, and then pushed to the device. This is the directive that the sample RSS reader uses. Finally, we reserve `DandC` in case the data structure might be updated by both the device and the cloud. Resolving potential update conflicts in general is tricky. Therefore, currently we do not allow DandC in SLIM. However, DandC could be useful in practice. For example, it is needed if we model an email folder as a vector of individual email messages, where user can delete emails while newly received emails may be added by the cloud. We are currently experimenting with different designs and the DandC scheme may be supported in future work.

- `Consistency` gives a few options as when the updates should show up at the other end. Currently we support three options: `OnTimeBound` and `OnDirtyBound` are bound by refreshing cycles and number of dirty items, respectively. Changing the consistency directive in the above sample code to `OnTimeBound` with the parameter `freshInterval` would make our RSS reader refreshing in the same manner as the STL-based RSS reader. Finally, `OnDemand` simply forces the update at user's chosen point.

These set of directives transform the stateless implementation of the RSS reader into an event-driven one, where the logic to display the feeds is triggered by incoming new items. Notably, the logic of parsing the XML object is no longer needed on the device side. This logic is instead residing on the cloud side. The cloud will perform the parsing and generation of the native vector that is now mapped onto the device. This leads to both network and computation savings (Section 4) with little additional programming effort.

Multiple data structures can be linked together to enable rich functionality. Indeed, a generic mailbox will not only include the main vector to contain email bodies but also multiple sorted vectors on various attributes, as well as index that allow search and lookup. In the RSS example, the feeds can derive an index in the form of a map at the cloud, and we can then map it into the client, allowing keyword-based search:

```
typedef SLIMVector<PolyPointer<FeedItem>, L3, C2D, OnDirtyBound
    <1> > FeedIndex;
//cloud—built inverted index on words, derived from the feeds vector
SLIMMap<string, FeedIndex, L3, C2D, OnDirtyBound<1> > index("
    RSS URL/index");
Promise<FeedIndex> resultPromise = index[key];
resultPromise.continueWith([] (FeedIndex result) {
    //display the search result for the key
    foreach (feed in result)
        display(*feed);
}).ignore();
```

The code snippet above relies on an L3 inverted index that is built at the cloud, each entry of the index contains pointers to the feeds containing the keyword in question. The code contains two new notions. The first is *PolyPointer*, which is a mechanism to allow pointer access across memory hierarchies (including the cloud), and will be explained in detail in Section 3. The second is *promise* [5].

The rationale of using promise here is that including cloud as a memory hierarchy introduces the unreliable network connectivity. Completely hiding this from programmer is impossible, as this moves too much complexity into the runtime. One significant complexity stems from the fact that, since SLIM is a family of generic data structures, its instance can contain pointers. As such it is possible that the target of the pointer is absent from the device and reside in the cloud at the time of pointer dereferencing. To alleviate this problem, we borrow the interface of promise to deal with asynchronous programming.

An instance of promise represents a future result value from an operation. Initially, the promise is returned by certain operation immediately (such as the `operator[]` in our case) in the *unresolved* state, expecting to receive a result at some unspecified future time. When the result is received, the promise becomes *fulfilled* and the result becomes the value of the promise. Promise will then call the continuation routine bound by the `continueWith` interface. In the above case, we bind a lambda function to deal with the resolved result value. If an error occurs, either in the calculation or network transmission, the promise becomes *broken*. User can either attach an exception routine to handle the error or ignore the exception, as we have done with the `ignore()` interface in the example code. Note that if the entries of the index, as part of the L3 SLIMMap, are available locally, the query will be satisfied immediately.

# 3. SYSTEM DESIGN

This section focuses on the design of the SLIM vector. Just like that of a STL vector, an element in a SLIM vector can be any fixed size data, and can even contain pointers. We omit the details of L1 SLIM vector, as it is simply a regular STL vector that provides an in-memory dynamic array of data items.

Even though we use vector as a representative data structure to describe SLIM, SLIM actually supports a wide variety of other generic data structures. Data structures such as map and linked list can be easily implemented on top of vectors since vector can be simply regarded as a continuous block of memory. Data structures build on top of a vector can also inherit the same set of policies of the base vector data structure they are based upon. However, this is often not optimal or even correct. For example, when a new item is inserted into a linked list that has a consistency policy of 1-dirty bound, we should send the update after the entry itself and both its previous and next entries are updated.

## 3.1 L2 SLIM vector

An L2 SLIM vector follows the external-memory model [13], in that its total size can be greater than its allocated memory. The user of the L2 vector doesn't need to explicitly manage memory space, with the trade-off of some performance overhead. Our goal is to maximize the performance such that the operations over the L2 vector approaches that of L1 when hit memory.

SLIM vectors are logically laid out in a non-overlapping 64bit *universal address* space. A SLIM vector occupies a continuous range in the space, logically with a starting pointer and an end pointer. Opaque to the programmer, a SLIM vector is located with an internal root pointer. The root pointer must be persistent and available during the entire lifetime of the vector. An L2 vector is given a pool of memory pages at its birth, but that pool may shrink or grow transparent to the programmer.

The most significant complexity arises because of pointers: a user can instantiate a pointer to walk through the vector in arbitrary fashion. Likewise, internal operations such as push_back relies on pointer as well. An L2 vector, by definition, is not guaranteed to have all of its elements memory resident.

For efficiency reason, a pointer in SLIM should contain the in-memory address of the point-to object (in our case, another SLIM vector) if the pointed-to object is in memory. Like normal C++ pointer, the content of the L2 pointer should allow direct access to memory address, instead of relying on extra translation of the logical universal address each time. The process of translating a universal pointer to a local address is commonly known as *pointer swizzling*. Similarly, *unswizzling* is the reverse process to make sure that direct access is disallowed, when the point-to object has been removed from memory. Enabling these two processes takes extra bookkeeping. There is a rich body of literature on pointer swizzling, we refer reader to [9] which classifies different proposals along the dimensions of the timing when swizzling occurs and the behavior when an object is displaced from memory.

A pointer in SLIM is a *PolyPointer* which, along with other meta data such as size and capacity of the vector it points to, includes two fields, UA and MA, for universal address and physical memory address, respectively. The state machine is depicted in Fig. 1, assisted by two auxiliary SLIM-internal data structures. The first is the PPT (PolyPointer Table) that records the MA-to-UA mapping, with the granularity of a page (typically 4KB). Given a memory address, PPT allows direct inspection of the UA page of the corresponding memory page. The second data structure is RHT (Reverse Hash Table) that enables the reverse lookup, returning the MA for a given UA. These two tables are always updated atomically to ensure their mutual consistency. RHT also records the disk location for pages that are not swapped out. RHT is persistent, whereas PPT can be derived from RHT.

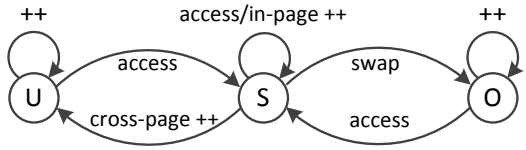


**Figure 1: PolyPointer state machine. Pointer primitives that may cause state transition include self-increment (++), dereference(access) and swapping. Also an in-page self-increment is also distinguished from a cross-page one.**

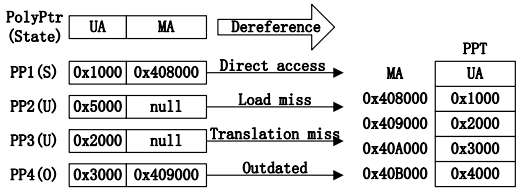


**Figure 2: Four possible events upon the dereference of PolyPointers. Dereferencing unswizzled PolyPointers (its MA field is null) will raise a translation miss to fill the MA or a load miss if the UA is also not available. Swizzled PolyPointer will have a registered UA value in the PPT, which matches its MA field. To identify an outdated PolyPointer, SLIM checks if the UA field equals the value stored in the PPT entry that maps the MA.**

A new PolyPointer is initialized with a universal address, but its MA is NULL. Such a PolyPointer is in state *Unswizzled* (shown as "U" in Fig. 1), dereferencing such a pointer causes a translation miss. Handling such misses can follow a few different paths. Assuming that the allocated memory has been used up. One of the used pages will be evicted, written to disk if dirty. The PPT entry now records UA of the dereferenced PolyPointer, the RHT is updated, and the MA field is calculated from the PPT entry position and the offset in the page (Fig. 2). It is also possible that another PolyPointer within the same UA page suffers a miss as well. In this case, the PPT entry already exists, and the only thing needs to be done is to update the MA field of that PolyPointer. In this and the earlier case, the pointer now becomes *Swizzled* (shown as "S" in Fig. 1). A Swizzled PolyPointer can become Unswizzled, this happens if it steps beyond a page boundary. The next dereferencing will suffer a translation miss (and a possible load miss).

If a target memory is replaced, the corresponding PPT entry will record the UA of a different PolyPointer. All PolyPointers whose MAs point to this entry without the matching UA are now in the state *Outdated* (shown as "O" in Fig. 1). There are several approaches to discover and fix a PolyPointer in this state. For instance, the state can be explicitly changed to *Unswizzled* at the time of replacement, requiring either a reverse mapping table or scanning.

In the current implementation, we leave the pointer unchanged, but require consulting PPT at the time of dereferencing the pointer. This strategy works well if vector contains a large amount of data in its body, so that such overhead is amortized. Handling load miss is straightforward, RHT is first consulted with the requesting UA to locate the page off the stable storage, and PPT is then updated to record the memory page that the page is loaded into.

We set an upper limit on the secondary storage that can be used by SLIM. Of course, potentially Flash may fail or may run out of space. We handle these by throwing exceptions, similar to traditional STL when it run out of memory.

## 3.2 L3 SLIM vector

We model L3 as a pair of synchronized L2 vectors, one at the device and one at the cloud. To keep it simple, our current design focuses on C2D and D2C, in which cases *application updates* happen only at one end of the pair, and are reflected to the other end with *SLIM updates*, based on its consistency policy. An update expands the *view*. The one receives application updates is called the *master view*, whereas the one receives updates is called the *slave view*. For the time being we do not consider the shrinking of a view (i.e. delete) by simply tag the deleted entry rather than removing it.

Updates as well as load miss at the device must be resolved with the same universal address. Therefore in L3 the management of the universal address space is in the cloud, independent of where the addresses are consumed. In the case of D2C, the device will ask a chunk of space from the cloud, and then assign them at the device when it receives application updates. The corresponding SLIM update will include the universal address that was applied in the device.

Handling SLIM updates can take different approaches. Our current implementation records application updates in a log, and ship the log to be replayed at the other end. An alternative is to perform updates through a shared key-value store, where multiple updates can be merged. Some scenarios may require the instantiation of a new SLIM vector in one shot. For instance, if a vector is to be sorted, it is easier to derive the sorted vector in the cloud and map it to device while destroying the old one, than updating the entries as the sorting is progressing.

We require all data in the view to be available at the cloud. This is justified because storage is cheap in the cloud, and some of the data needs to be kept indefinitely anyway (e.g. gmail). Since cloud has all the data, it is possible for the device to freely reclaim storage resources as it sees fit. The only additional bookkeeping is to mark in RHT that the page is now accessible only from the cloud. When the device suffers a load miss, the universal address enclosed in the request allows the cloud to retrieve the data.

Cloud may issue update events when the device is accessing the data structure and vice versa, thus causing race conditions. SLIM allows user to specify policy to coordinate concurrent operations when update arrives while the user is accessing the data. The user can choose a more restricted access policy using critical sections, or more relaxed policy as long as the application semantic can tolerate benign races. One such case is when the application loops through the vector while SLIM updates adds new elements at the tail.

# 4. EVALUATION

In this section we report some initial results of a prototype SLIM implementation. We first use a micro benchmark to evaluate the performance of SLIM vector when no network traffic is incurred, to validate our pointer swizzling algorithm and measure its overhead. We then conducted a case study using RSS reader as an example to demonstrate the advantage of SLIM when applied in real applications. All the experiments are performed on a Windows 7 machine with 2.13GHz Intel Core2, 2GB main memory, and 1Gb ethernet.

## 4.1 Microbenchmark

We measured the performance of SLIM vector under microbenchmarks with sequential read/write, random read/write, and push_back operations. We compared it with the performance of regular STL vector under the same microbenchmarks. The push_back benchmark appends elements one by one to the tail of the vector until the total data size reaches 512KB. Other benchmarks perform 512K operations, each for one element, on an initialized vector with 512KB of total data. In the experiments, we configured SLIM vector to be L3. In order to avoid network traffic during the measurement, all the data in the vector were touched once to prime the data into main memory, and we set the consistency policy to OnDemand so that updates will not propagate to the cloud.

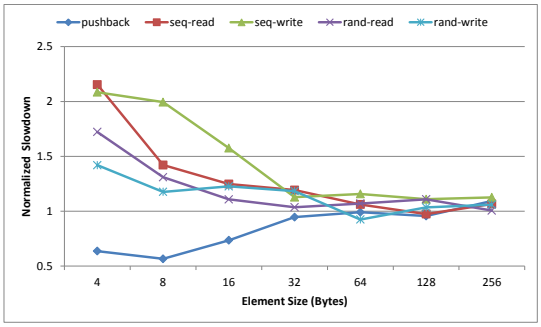


**Figure 3: Microbenchmark results on SLIM vector primitives, normalized to regular vector performance.**

Fig.3 reports the normalized performance of SLIM vector relative to that of regular STL vector. The x-axis represents the element size of the vector in bytes. As expected, SLIM vector generally performs worse than STL due to the inherent overhead[1]. As shown in the figure, the larger the element size, the smaller the performance difference between SLIM and STL. As elements get larger, the cost of the operations on the payload becomes larger, and hence the relative cost for maintaining the states of the SLIM metadata becomes smaller. When element size is larger than 32 bytes, the performance difference between SLIM and STL vectors ($\sim$5%) is no longer significant. In many real world applications that share data between cloud and devices, element size is often relatively large (e.g. an RSS feed, or an email message). We believe that for many such real world scenarios, the overhead of SLIM vectors relative to STL is insignificant.

## 4.2 Case Study on RSS Reader

We implemented an RSS reader prototype using SLIM vector. It essentially is a traditional RSS reader partitioned into a cloud part and a device part. The cloud part fetches XML files from the original RSS content provider, parses them into feed items and stores the feeds in a SLIM vector. The device part directly accesses the SLIM vector to get the updated RSS feeds. There are two major benefits for RSS reader implemented this way: 1) the job for parsing XML files is migrated to the cloud, which saves energy in the device side; 2) the network traffic is also greatly reduced because there is no duplicated feed items in the SLIM vector, and hence only the newly updated feeds need to be transferred. In contrast, a traditional RSS reader needs to fetch the entire XML file, which often contains old feed items.

To measure the computation savings, we profiled the execution of XSD [2], an open source XML parser written in C++, for parsing an XML file from Yahoo News [3] which contains 20 feed items. The parsing task takes about 24 million processor instructions. The total overhead is proportional to the number of times the feed is redundantly transfered in the stateless implementation.

[1] An exception is the push_back benchmark, which is caused by more aggressive function inlining by the compiler for SLIM

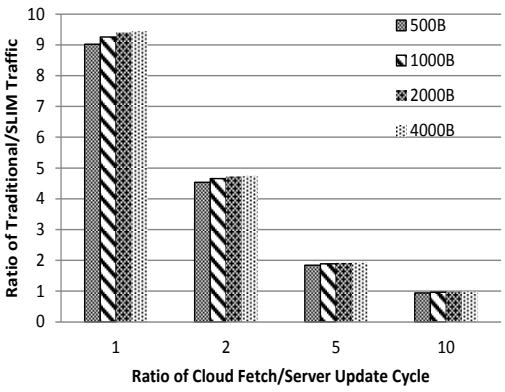


**Figure 4: Comparison of network traffic between SLIM and normal RSS reader. The traffic is normalized to SLIM reader. The feed item size varies from 500B to 4000B**

We also evaluated the device traffic saving compared to the traditional RSS reader. In the experiment, we built an RSS provider that maintains 100 channels and produces new feed items for each channel periodically and notifies the subscriber. Each channel organizes the latest 10 feed items in an XML file. For SLIM RSS reader, we measured the network traffic between the device and the cloud, while for the traditional one, we just measured the network traffic between the SLIM cloud service and the RSS content provider, since the cloud service itself requires the same traffic as the traditional RSS reader.

Fig. 4 shows the results. We change the number of newly generated feed items that triggers one notification to the subscriber (as shown in the x axis). The y axis represents the ratio between the network traffic of the traditional and the SLIM RSS reader.

In most cases, SLIM RSS reader's network traffic is much smaller than the traditional one. This is because SLIM breaks down the granularity of network data transfer from XML file to feed item, thus device only needs to fetch new feed items. In particular, if the cloud fetches the XML file every time when a new feed is generated, the traditional RSS reader's network traffic is more than 9 times of SLIM's. On the other extreme, when cloud service fetches XML file only when all its contents are updated, the traffic is about the same because SLIM RSS reader's meta-data traffic overhead is no more than the traditional RSS reader's update notification traffic.

# 5. RELATED WORK

Mobile computing leveraging cloud has been a extremely popular topic, and we can only discuss a few works with limited space. One dimension to compare is the granularity and partition point of device/cloud computation. Existing proposals include virtual machine (Internet Suspend/Resume [1] and CloudLet [11]), method-/procedure call (Cyber foraging [4], CloneCloud [6] and MAUI [7]), and explicit event such as tuple space [10]. SLIM adds to this landscape with the data structure angle.

Extending memory hierarchy beyond the single main memory hierarchy is an old idea. Still, the external memory model [13] and existing implementations such as STXXL [8], usually include secondary storage only. Our design leverages and extends the pointer swizzling technology across all three levels of memory hierarchy, transcend the device and cloud boundary. We also borrow the latest advancement of asynchronous programming to deal with network unreliability.

# 6. DISCUSSION AND FUTURE WORK

Instead of building a monolithic application, SLIM requires the application to be partitioned into a cloud and a device parts which interact with each other by sharing the SLIM data structures. In our prototype, the partition is done manually. For future work, we are looking at building a SLIM-aware compiler that can automatically partition the code with the help of some user annotations.

Currently, SLIM is implemented as a library and is linked into the applications. Therefore, its allocation policies are managed in a per-application basis. It is potentially beneficial to build SLIM as a middleware and being managed by the device OS. The flexibility of freely modifying memory allocation transparently to the application affords the possibility of controlling and prioritizing memory usage in mobile device depending on application states (e.g. foreground or background) and system requirements (e.g. shutting down part of the main memory to get into low energy mode).

In this paper, we assume the cloud is reliable and it holds the "golden" copy of the application state. In reality, a machine in the cloud can fail, and the process in the cloud may lose its in-memory state. It is important that the clould part can be restarted and quickly recover to a consistent state with the device. For future work, we plan to add automatic logging of SLIM data structure updates to the cloud. The log needs to be stored in a reliable cloud file system so that individual machine failure will not cause the loss of user data.

Our initial evaluation is encouraging. However, significant work remain to fully demonstrate the power and validate the vision. To be pragmatic, we plan to examine the design of a few demanding applications, including browser, file system and mailbox, each of which may contain multiple data structures. This exercise can then guide the building of the SLIM library. At the system level, an interesting direction is to install policies to control the footprint of various data structures so as to minimize energy consumption.

# References

[1] The internet suspend/resume project. http://isr.cmu.edu/.

[2] Xsd. http://www.codesynthesis.com/products/xsd/.

[3] Yahoo news. http://rss.news.yahoo.com/rss/topstories.

[4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. i Yang. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop*, pages 87–92. ACM Press, 2002.

[5] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: A Framework for Cloud Computing. Technical report, Technical Report MSRTR-2010-159, Microsoft Research, 2010.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th European conference on Computer systems*, EuroSys '11, New York, NY, USA, 2011. ACM.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[8] R. Dementiev and L. Kettner. Stxxl: Standard template library for xxl data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

[9] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. *The VLDB Journal*, 4:519–567, July 1995.

[10] A. L. Murphy, G. P. Picco, and G. catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15:2006, 2006.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, October 2009.

[12] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), HP Laboratories, 1995.

[13] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33:209–271, June 2001.

# A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity

Animesh Trivedi
IBM Research
Saeumerstrasse 4
Rueschlikon, Switzerland
atr@zurich.ibm.com

Bernard Metzler
IBM Research
Saeumerstrasse 4
Rueschlikon, Switzerland
bmt@zurich.ibm.com

Patrick Stuedi
IBM Research
Saeumerstrasse 4
Rueschlikon, Switzerland
stu@zurich.ibm.com

## ABSTRACT

Modern cloud computing infrastructures are steadily pushing the performance of their network stacks. At the hardware-level, already some cloud providers have upgraded parts of their network to 10GbE. At the same time there is a continuous effort within the cloud community to improve the network performance inside the virtualization layers. The low-latency/high-throughput properties of those network interfaces are not only opening the cloud for HPC applications, they will also be well received by traditional large scale web applications or data processing frameworks. However, as commodity networks get faster the burden on the end hosts increases. Inefficient memory copying in socket-based networking takes up a significant fraction of the end-to-end latency and also creates serious CPU load on the host machine. Years ago, the supercomputing community has developed RDMA network stacks like Infiniband that offer both low end-to-end latency as well as a low CPU footprint. While adopting RDMA to the commodity cloud environment is difficult (costly, requires special hardware) we argue in this paper that most of the benefits of RDMA can in fact be provided in software. To demonstrate our findings we have implemented and evaluated a prototype of a software-based RDMA stack. Our results, when compared to a socket/TCP approach (with TCP receive copy offload) show significant reduction in end-to-end latencies for messages greater than modest 64kB and reduction of CPU load (w/o TCP receive copy offload) for better efficiency while saturating the 10Gbit/s link.

## Categories and Subject Descriptors

C.2.5 [**Local and Wide-Area Networks**]: Ethernet; D.4.4 [**Communications Management**]: Network communication—*RDMA, Efficient data transfer, performance measurement*


*APSys* '11 Shanghai, China


## General Terms

Design, Measurement, Performance

## Keywords

RDMA, Commodity Clouds, Ethernet Networking

## 1. INTRODUCTION

Applications running in today's cloud computing infrastructures are increasingly making high demands on the network layer. To support current data-intensive applications, terabytes worth of data is moved everyday among the servers inside a data center requiring high network throughput. For large scale web applications like Facebook low data access latencies are key to provide a responsive interactive experience to users [4]. And recently there have been cases running HPC workload inside scalable commodity cloud strengthening the need for a high performance network stack further [1]. To accommodate those requirements modern cloud infrastructures are continuously pushing the performance of their network stack. Most of the work focuses on improving the network performance inside the virtualization layers [14, 21]. In parallel some cloud providers have upgraded parts of the networking hardware to 10GbE which recently has moved closer to the price range acceptable for commodity clouds.

While high network performance is greatly desired, it also increases the burden for end hosts. This is especially true for applications relying on standard TCP sockets as their communication abstraction. With raw link speeds of 10Gbit/s, protocol processing inside the TCP stack can consume a significant amount of CPU cycles and also increases the end-to-end latency perceived by the application. A large part of the inefficiency stems from unnecessary copying of data inside the OS kernel [3]. The increased CPU load is particularly problematic as parallelizing the network stack remains challenging [25].

High performance networks are state of the art in supercomputers since a long time. For instance, the Remote Direct Memory Access (RDMA) network stack provides low-latency/high-throughput with a small CPU footprint. This is achieved in RDMA mainly by omitting intermediate data copies and offloading the protocol processing to hardware [16]. Adopting RDMA by commodity clouds is, however, difficult due to cost and special hardware requirements. In this paper we argue that the basic concept and the semantics of RDMA can be beneficial for modern cloud infrastructures

[1] http://aws.amazon.com/hpc-applications/

as the raw network speed increases, and – most importantly – that RDMA network support can efficiently be provided in pure software, without a need for special hardware. To demonstrate our findings we have developed SoftiWARP, a software-based RDMA stack. We have evaluated our stack against a socket-based approach and show that SoftiWARP provides high-performance network I/O with reduced CPU load and significantly lower end-to-end-latencies for reasonably large message sizes.

# 2. REVISITING SUPERCOMPUTER NETWORKING

Supercomputers are designed to run specialized workloads and are built using customized hardware and interconnects. The type of workload running on supercomputers, e.g. MPI-based data processing, requires super-fast and efficient network I/O to shuffle data across the interconnects. At a first glance, it seems apparent that the requirements faced by supercomputers in the past resemble the ones of future commodity data centers. Thus, a natural next step would be to look at the network stack operated by supercomputers and see if a similar approach could be beneficial for data centers as well. RDMA is one networking technology used by supercomputers. RDMA enables applications to have high bandwidth and low latency access to data by efficiently moving data between application buffers. It is built upon the *user-level networking* concept and separates data from the control path, as only the latter requires host OS involvement. An important aspect of RDMA when compared to socket-based networking is that RDMA has rich asynchronous communication semantics, that helps in overlapping communication with computation for better resource utilization.

## 2.1 Commodity Clouds and RDMA

Clouds run on data centers built from commodity hardware and perform on-demand, flexible and scalable resource allocation for the applications. Resource utilization becomes an important factor to cloud efficiency because hardware resources (e.g. CPU, memory, and networks) are being multiplexed among many virtual machines. Due to the economy of scale, inefficiencies in resource usage can potentially eclipse large gains from the clouds. In the following we look at CPU usage, latency and energy consumption in data centers and discuss how RDMA can improve the performance in those cases.

**CPU Usage:** Network I/O can cost a lot of CPU cycles [3]. Applications in clouds require better support from the OS to efficiently transfer large quantities of data. RDMA-enabled NICs (RNICs) have sufficient information to completely bypass the host OS and directly move data between application buffers and the network without any intermediate copies. This requires considerably lower CPU involvement and frees up CPU cycles for productive application processing.

**Latency:** Low data access latencies are key for large-scale web applications like Twitter or Facebook in order to provide sufficient responsiveness to user interactions [20]. Low data access latencies also play an important role in determining which consistency model a cloud storage layer can possibly implement [23]. And finally, absence of timely access to the data may render certain applications inefficient and cause a loss of productivity. RDMA helps in reducing end-to-end application latencies as it does not require local or remote applications to be scheduled during network transfers. As a consequence RDMA also is minimizing the penalty for processor state pollution such as cache flushes.

**Energy:** The energy consumption of commodity data centers have been alarming [2]. Future cloud infrastructures need to squeeze more performance per Watt. Because of an efficient data movement pattern on the memory bus (zero-copy), less application involvement, and low OS overhead, RDMA data transfers are more energy efficient than traditional socket based transfers [12].

## 2.2 Challenges for RDMA in Clouds

Although promising, RDMA has neither extensive end-user experience and expertise nor widespread deployment outside the HPC world. We point out three major challenges in deploying RDMA hardware in commodity cloud infrastructure.

First, to efficiently move data between NIC and application buffers, current RDMA technology offloads the transport stack. The issues related to integration of stateful offload NICs in mature operating systems are well documented [16]. These problems include: consistently maintaining shared protocol resources such as port space, routing tables or protocol statistics, scalability, security and bug fixing issues etc.

Second, clouds run on commodity data centers. Deploying RDMA in such an environment might be expensive. RDMA requires special adapters with offload capabilities like RNICs. The usability of those adapters, however, is limited to a couple of years at best since processor and software advancements are catching up rapidly.

Third, RDMA technology has often been criticized for its complex and inflexible host resource management [10]. RDMA applications are required to be completely aware of their resource needs. System resources such as memory, queue pairs etc. are then statically committed on connection initialization. This goes against the cloud philosophy of flexible and on-demand resource allocation.

# 3. RDMA IN SOFTWARE

Despite challenges we argue that a major subset of RDMA benefits still can be provided to commodity data center environment using only software-based stacks. In this section we present a case for the software-based RDMA stack, discuss its advantages, and why it is a good match for the clouds.

First, there are currently several networks stacks supporting RDMA communications semantics. iWARP [11] (Internet Wide Area RDMA Protocol) being one of them, enables RDMA transport on IP networks. In contrast to other RDMA stacks like Infiniband, iWARP uses TCP/IP or SCTP/IP as end-to-end transport. This enables RDMA on the ubiquitous Ethernet infrastructure typical to commodity data centers and opens the door to inexpensive, Internet-wide and pure software-based implementations.

Second, we believe that the availability of sufficient application specific knowledge (passed using the RDMA API) during network protocol processing can single-handedly boost efficiency [6], obsoleting offloading in many cases. The rich RDMA API supports one-sided pull/push operations, semantic aggregation of operations, grouping of work postings, and completion notifications etc. Using this knowledge with modest support from the OS, goals of the RDMA stack such as zero copy transmission, direct data placement into appli-

cation's buffer, and one-sided operations are possible even in a software-based solution. This results in better end-to-end latencies, and low CPU footprint in the commodity data centers.

Third, with its flexible resource management policies, a software-based solution is matching closely with the requirements of the cloud. It also helps in materializing *on-demand RDMA networks* between hosts. While dynamically managing resources can result in some performance loss, applications can amortize this cost up-to an extend by reusing I/O critical resources such as memory regions. Please note that raw performance is not the primary concern for such a software-based stack. It can not outperform a hardware based solution but still can provide a large subset of RDMA benefits in pure software.

## 3.1 SoftiWARP

We have implemented the iWARP protocol in a software stack for Linux called *SoftiWARP* [15]. SoftiWARP enables commodity Ethernet NICs to handle RDMA traffic in software. It aims to provide the benefits of rich RDMA semantics, while avoiding most of the problems with TOEs or RNIC deployment. By integrating transparently within industry standard OFED stack [18], SoftiWARP appears to the application as a regular RNIC device. It also interoperates with available RNIC hardware such as the Chelsio T3 adapters.

### 3.1.1 Comparing SoftiWARP with plain TCP

Historically, TCP implementations have been closely associated with the BSD socket interface. The socket API abstracts all protocol details and exports a simple send and receive interface for data transfer. From an applications perspective, all additional communication semantics must be a part of the application itself. With this most of the data path optimizations RDMA is focusing on are not consistently achievable, despite some attempts to augment the OS networking stack with shortcuts depending on implicit state information(e.g. [5]). In this section we highlight the advantages RDMA brings to applications and compare it to socket based TCP network I/O operations:

Application Data Copy Avoidance: To preserve system call semantics, the socket interface must copy application data between user space and kernel. The RDMA API helps to avoid copy operations on the transmit path by explicitly passing buffer ownership when posting a work request. Any sending or retransmission of data can be done directly from the user buffer. However using kernel TCP sockets, the current SoftiWARP implementation does not explicitly know when data is finally delivered to the peer and restricts zero copy to process a non-signaled work. A simple check of TCP send state information would be sufficient to appropriately delay work completion generation.

Less Application Scheduling: With the necessary data placement information known, SoftiWARP places data directly into user's buffers from the TCP receive softirq processing context. Also, while processing one-sided operations, it does not schedule the application for network I/O. These characteristics are particularly attractive for applications such as media streaming, CDNs, distributed data storage and processing etc. Furthermore, work completion processing is minimized to the application needs – completing non-signaled work will not trigger application scheduling.

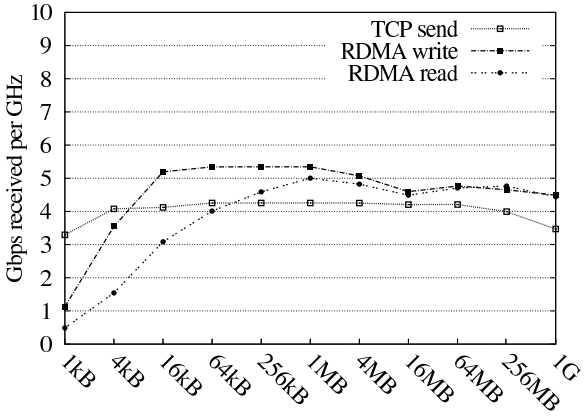


**Figure 1: Receive efficiency of TCP w/o receive copy offloading and SoftiWARP**

Non-blocking Operations: The asynchronous application interface frees applications from blocking and waits on data send or receive operations to complete. If semantically needed, the application may still do a blocking wait for completion.

Flexible Memory Management: User buffer pinning in memory is inevitable for hardware RDMA implementations [10]. However a software based in-kernel RDMA implementation has flexibility of performing on-demand buffer pinning, once source or sink address and data length is known. This puts memory resource economics on par with kernel TCP stack, though it may result in some performance loss.

### 3.1.2 SoftiWARP and RNICs:

Looking beyond the context of this paper, SoftiWARP will likely not replace RNICs on given installations. It will also not obsolete their applicability on high-end server systems or if very low delay requirements are stringent. Rather, it enables RDMA communication on any commodity system. Heterogeneous setups, deploying RNICs on the server side and running a software RDMA stack on the typically less loaded client, are thus made possible.

# 4. PERFORMANCE

In this section we will give some early performance results for SoftiWARP. The numbers reported are average of 3 runs, each lasting 60 seconds. The experiments were run on identical IBM[1] HS22 blades containing dual Quadcore 2.5 GHz Intel Xeon CPUs (E554), 8GB RAM and connected using Mellanox ConnectX 10GbE adapters. All experiments were done on Linux (`2.6.36`) using netperf [17] with our extensions for RDMA tests and oprofile [19]. We plan to open source these extensions which consist of uni-directional data transfer tests. To obtain CPU usage, we divided the total number of CPU samples obtained during a benchmark run by the number of samples obtained when we ran a busy loop on a single CPU. Hence they include the overhead of buffer allocation and page pinning for the RDMA. We report two variants of TCP performance, with and without TCP receive copy offload using Intel QuickData [1]technology in Linux(`CONFIG_NET_DMA`). Although helpful with the CPU usage for small size messages, in our experiments we have found TCP receive copy offloading to be major source of performance degradation for the large message sizes.

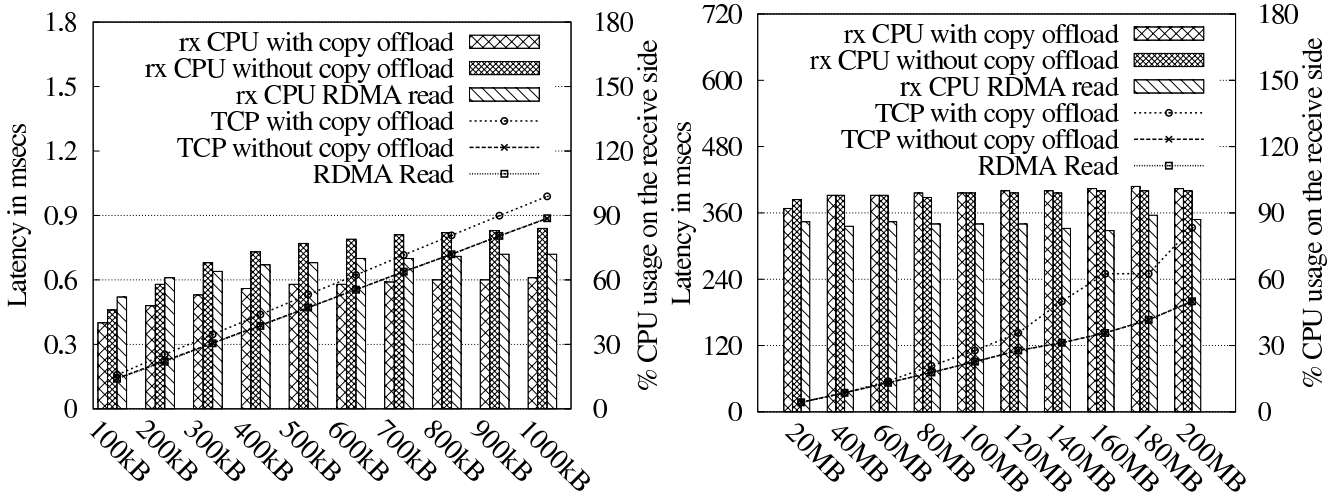


**Figure 2: Round trip latencies for TCP request response and RDMA read a) 100-1000kB, b) 20-200MB data blocks**

**Bandwidth:** SoftiWARP does not impose any performance penalty for large buffer sizes. Compared to TCP's buffer size of 2kB, SoftiWARP achieves full line speed with buffer sizes of 16kB and 32kB for RDMA Write and Read tests respectively (not shown in the paper). If aggregate buffer posting using multiple work elements (100) is performed, it saturates the link with message sizes of 8kB and 16kB for RDMA Write and Read tests respectively. We are working on optimizing SoftiWARP's performance for transmitting small buffer sizes.

**CPU Efficiency:** SoftiWARP is very economical with the CPU cycles. By using zero copy mechanism on the transmit side, SoftiWARP's CPU usage for non-signaled workloads is approximately 40%-45% (not shown), which is less than a send call and at par with TCP sendpage mechanism. However for, small buffer size (less than 64kB) send is more efficient. On the receiving side SoftiWARP consumes 8%-22% less CPU than TCP while receiving at the full link speed. Figure 1 shows the receive side efficiency of SoftiWARP and TCP (w/o TCP receive copy offload) in terms of Gbits received per GHz of CPU use. Beyond a modest 16kB (64kB for reads), SoftiWARP's receive is more efficient than the TCP receive.

**Latency:** The latency experiment consists of a request (4 bytes) and response (variable, asked size) system. Figure 2 shows end-to-end delay of a request with the CPU usage on the receiving side. For small requests (in kBs), TCP copy offloading saves significant CPU cycles (13%-27% compared to the non-offloaded version) but it results in a non-negligible performance penalty for the end-to-end latencies. SoftiWARP offers 11%-15% (for 100kB-1000kB range) and a significant 5%-67% (20MB-200MB range) reduction in end-to-end application latencies over TCP (copy offloaded version). However without TCP receive copy offloading, the performance gap closes at a cost of higher CPU usage (5%-18%). On the response transmitting side by using in-kernel sendpage mechanism for already pinned pages, SoftiWARP improves the CPU usage by 34%-50% (not shown) against TCP.

The CPU and performance gains are primarily achieved by avoiding application involvement and exploiting zero copy mechanism for the transmission. We will provide an in depth-analysis of the system in future.

## 5. RELATED WORK

iWARP is built upon the Virtual Interface Architecture (VIA) concept, which has roots in systems such as U-Net [24], Hamlyn [8], ADC [9], Typhoon [22] etc. Although these systems helped in developing the key principles behind iWARP and RDMA, but in the absence of any standardization work lead to multiple ad-hoc implementations of application and network interfaces. SoftiWARP is an effort to find a golden middle ground between trivial read/write system calls on sockets and exotic user-accessible network interface exports.

Unlike true user-level networking systems, SoftiWARP does not run the network stack in user space. Networking remains to be a service of the OS but user-tailored. SoftiWARP uses application specific knowledge, which is available through rich RDMA API, to make efficient use of resources. Also, giving resource control to the kernel instantiates its status as a resource arbitrator that helps in imposing global policies [13]. The Ohio Supercomputer Center has presented another kernel based iWARP implementation [7]. Their system, however, is not compatible with the OFED framework.

## 6. CONCLUSION

With the availability of 10Gbit/s Ethernet for commodity data centers and the ever increasing demand of distributed applications for efficient inter-node network I/O, network stacks are challenged to keep up with the pace. In this paper, we argue that traditional socket-based networking is not capable of satisfying those future demands, as it puts too big of a burden onto the host. Instead we propose the use of RDMA as a networking concept to overcome those issues. RDMA traditionally has been used as a high-performance networking technology in supercomputers, but they are costly and require special hardware which makes them difficult to be adopted by commodity data centers. However, we believe that the main advantages of RDMA can be provided in pure software inside commodity data cen-

ters. In the paper, we discuss an early implementation of a software-based RDMA solution and present experimental results. The results confirm the high-throughput performance of such approach while significantly reducing end-to-end application latencies than plain socket based TCP/IP.

In the future we plan to investigate potential performance benefits of SoftiWARP in the presence of OS and application level virtualization. SoftiWARP integrates seamlessly with asynchronous communication APIs found in, e.g., Java NIO, and may allow to efficiently bypass virtual layers down to the hypervisor.

SoftiWARP is an open source project and the code is available at `www.gitorious.org/softiwarp`.

# 7. REFERENCES


[1] Intel Corporation. Intel QuickData Technology Software Guide for Linux* at `http://www.intel.com/technology/quickdata/whitepapers/sw_guide_linux.pdf`, 2008.

[2] D. G. Andersen et al. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SOSP*, pages 1–14, 2009.

[3] P. Balaji. Sockets vs RDMA interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *Proceedings of RAIT workshop*, 2004.

[4] D. Beaver et al. Finding a needle in haystack: facebook's photo storage. In *Proceedings of the 9th OSDI*, pages 1–8, 2010.

[5] H.-k. J. Chu. Zero-copy TCP in Solaris. In *ATEC '96: Proceedings of the 1996 USENIX ATC*, pages 21–21, 1996.

[6] D. D. Clark et al. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM*, pages 200–208, 1990.

[7] D. Dalessandro et al. Design and implementation of the iwarp protocol in software. In *Proceedings of IASTED*, pages 471–476, 2005.

[8] G. B. David et al. Hamlyn: a high-performance network interface with sender-based memory management. In *Proceedings of the Hot Interconnects III Symposium*, 1995.

[9] P. Druschel et al. Experiences with a high-speed network adaptor: a software perspective. In *Proceedings of ACM SIGCOMM*, pages 2–13, 1994.

[10] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *Proceedings of ICDCS*, pages 553–560. IEEE Computer Society, 2009.

[11] IETF. Remote direct data placement working group. `http://datatracker.ietf.org/wg/rddp/charter/`.

[12] J. Liu et al. Evaluating high performance communication: A power perspective. In *Proceedings of the 23rd ICS*, pages 326–337, 2009.

[13] K. Magoutis. The case against user-level networking. In *Proceedings of 3rd Workshop on Novel Uses of System Area Networks*, 2004.

[14] A. Menon et al. Optimizing network virtualization in xen. In *Proceedings of the USENIX ATC*, pages 2–2, 2006.

[15] B. Metzler, P. Frey, and A. Trivedi. SoftiWARP - Project Update, 2010. Available online at `http://www.openfabrics.org/OFA-Events-sonoma2010.html`.

[16] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th HotOS*, pages 1–5, 2003.

[17] Netperf, 2.4.5. `http://www.netperf.org/netperf/`, 2011.

[18] OpenFabric Alliance. OpenFabrics Enterprise Distribution (OFED) Stack, 2010. Available online at `www.openfabrics.org`.

[19] Oprofile, 0.9.6. `http://oprofile.sourceforge.net`, 2011.

[20] J. K. Ousterhout et al. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. In *SIGOPS OSR*. Stanford InfoLab, 2009.

[21] K. K. Ram et al. Achieving 10 Gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS VEE*, pages 61–70, 2009.

[22] S. K. Reinhardt et al. Tempest and typhoon: user-level shared memory. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 325–336, 1994.

[23] B. Tiwana et al. Location, location, location!: modeling data proximity in the cloud. In *Proceedings of the 9th ACM SIGCOMM HotNets*, pages 15:1–15:6, 2010.

[24] T. von Eicken et al. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM SOSP*, pages 40–53, 1995.

[25] P. Willmann et al. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the USENIX ATC*, pages 8–8, 2006.


## Notes

[1]IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Other product and service names might be trademarks of IBM or other companies.

# SPECTRE: Speculation to hide communication latency

Jean-Philippe Martin, Christopher J. Rossbach and Michael Isard
Microsoft Research, Silicon Valley, CA, USA

## ABSTRACT

We describe work in progress on the SPECTRE system which aims to provide high performance computing over distributed shared memory, targeting workloads such as graph algorithms for which functional or dataflow decompositions are inefficient. We exploit aggressive speculation to hide the latency of remote memory accesses and synchronization, and execute all code transactionally so that mis-speculations can be discovered and reverted. Unlike previous speculative transactional systems SPECTRE makes side effects visible beyond transaction boundaries before the transactions have committed, tracking dependencies to ensure correctness on abort: we call this property *transgression*. We outline the SPECTRE design and provide preliminary results from a microbenchmark to motivate the approach.

## 1. INTRODUCTION

This paper explores two linked hypotheses: that some algorithms are most naturally and efficiently implemented over a shared memory abstraction; and that speculation can be used to hide the communication latency that is frequently a bottleneck when implementing distributed shared memory.

We are building the SPECTRE system to test our hypotheses. It combines aggressive speculation with distributed software transactional memory to detect misspeculation and undo its effects. Our intention is to identify some performance-critical workloads that require more memory or processing power than is available on a single computer, but which benefit from the abstraction of mutable shared state. We will then compare their performance on our system with state of the art implementations on existing distributed execution engines such as MPI [18] and Dryad [12].

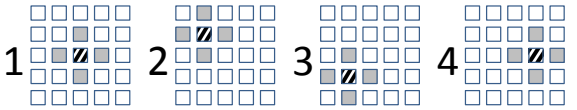


**Figure 1: Four, numbered, steps of a Loopy Belief Propagation algorithm. In each step four values (grey background) are read while a single value (striped) is written.**

Our focus on demonstrating the approach's potential leads us to building a runtime system first rather than an end-to-end programming model. The implementation and APIs of the system are designed to support high-performance general purpose distributed computing with speculation, and the applications it executes are ported by hand to exploit its features. We leave for future work some questions of developer simplicity, debugging support, and fault tolerance. The initial goal is to demonstrate that speculation can aid performance, and if this succeeds we plan to develop a layer that allows us to compile a high-level language directly to our runtime system.

The canonical type of algorithm that might benefit from SPECTRE is one that performs fine-grain updates to a large shared datastructure in an order that is difficult to predict statically. An example is sketched in Figure 1 which shows some steps in the execution of a Loopy Belief Propagation algorithm on a grid-graph: a small example graph is shown for clarity but a real application might span the memory of multiple cluster computers. In each step four values are read from the shared data-structure and used to compute a single result which is written back to the graph. The order of updates of graph nodes in this algorithm is a function of the previous updates so a strict functional decomposition introduces a dependency between each step; and since the ordering is data-dependent a static decomposition must conservatively insert a dependence between an update and the entire graph state at the previous step, which can be very inefficient. Many graph algorithms, including other machine learning and inference approaches, share this memory access pattern [16] as do standard problems such as mesh refinement.

In many practical cases like those shown in Figure 1

available parallelism is abundant, despite the fact that the algorithm is hard to express functionally. In the figure, the only constraint is that the operations in Step 4 must be executed after those in Step 1—Steps 2 and 3 could be executed in any order with respect to the other operations.

Speculation can help in two ways: first, some cores can run future steps in parallel, speculating that they do not need the output from the preceding steps. This allows a core to compute step 2 in parallel with step 1. Second, cores can advance through steps, speculating that what they have computed so far is correct. This allows a core to compute steps 1 and 4, consuming its own value without having to wait to hear whether the steps computed on other cores have modified what it is reading. In other words, speculation can discover parallelism and it can hide communication latency.

In order to maintain correctness it is essential to detect and undo mis-speculations. The engineering challenge we face is therefore to achieve performance gains from speculation that exceed the overhead introduced by bookkeeping and re-execution.

The SPECTRE system is currently a work in progress: we have a working prototype, but we are still exploring its design space, tuning its performance, and learning how best to program it. This paper therefore provides motivation and sketches the design, within the available space constraints, but reports only on provisional performance microbenchmarks.

## 2. COMPUTATIONAL MODEL

A SPECTRE program is made up of atomic code fragments called *tasks*. Instead of a total order as in a sequential program, one can specify a partial order of tasks to allow more parallelism. It can be represented as a *task graph*, as Figure 2 illustrates. If task $b$ is reachable from task $a$ in the task graph, then we write $a < b$. Our system allows a task to insert new successor tasks onto the graph during its execution, thus enabling iteration and recursion as in systems such as Cilk [1].

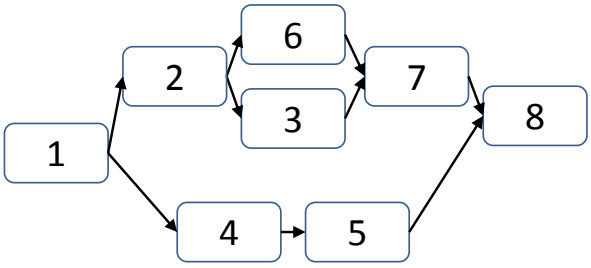


**Figure 2: The structure of a Spectre program.**

SPECTRE guarantees that the outcome of a correct program's execution is indistinguishable from a serial execution of the tasks that respects the partial order. The numbers in Figure 2 show one such possible order.

A task may receive read-only arguments when it is constructed, and all other inter-task communication is through a global set of shared objects. Each object has a unique identifier (OID), assigned at creation, which can be used as a reference to that object, for example by passing it as an argument to another task. Other side effects are possible, but for simplicity we focus on shared objects until Section 3.4.

The state of shared objects is managed using a transactional memory system in order to support aggressive speculation. Tasks may be executed concurrently or out of order, and consequent violations of the program serialization order are detected using the transactional memory. Mis-speculated tasks are then aborted and re-executed. For simplicity, and to maximize the opportunities for speculation, all code is executed transactionally.

In addition to adopting traditional software transactional memory techniques, SPECTRE makes side-effects visible beyond transaction boundaries when transactions have finished execution but not yet committed, a property we call *transgression*. This enables more aggressive speculation since there is no need to block computation while waiting for a distributed transaction commit. We track read dependencies between uncommitted transactions to maintain correctness in the face of transaction abort.

For the remainder of this paper, we use the term *transaction* to refer to an execution of a task. Each task may be executed multiple times, perhaps even concurrently, but at most one execution of a task will ever commit. The program completes when every task in the graph commits. We refer to transactions as though they were members of the partial order $<$. In addition we say a task $a$ has committed if and only if any execution of $a$ has committed.

## 3. SYSTEM DESIGN

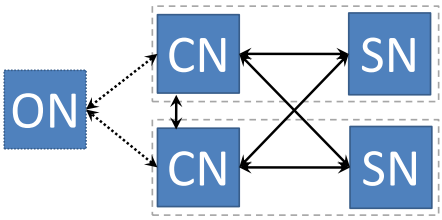


**Figure 3: The Spectre system**

The SPECTRE system (Figure 3) runs on a cluster of server computers and comprises a set of storage nodes (SN), a set of compute nodes (CN), and an ordering node (ON). These nodes are processes, and each computer may run more than one node: a typical deployment allocates an SN and a CN to every computer in the cluster. The SNs implement a distributed shared object space, the CNs execute transactions, and the ON coordinates task execution. The current implementation uses a centralized ON for simplicity, however we antic-

ipate this will become a bottleneck and a decentralized design to replace it is described in Section 5.

The SNs store the authoritative version of each object along with its version number. SNs can be queried for object versions, and they participate in the two-phase commit protocol described in Section 3.3. Each object is mapped to a unique, static SN. The SNs accept transactions in first-come, first-served order and have no knowledge of the task graph.

The ON stores the complete task graph, manages the assignment of tasks to CNs, and informs CNs when it is safe to commit a given transaction. When a SPECTRE application creates a task, it may specify a CN assignment for that task, or it can leave it up to the system to decide placement. Newly created tasks are sent to the ON, and it is informed when transactions commit.

Since the SNs are oblivious to the partial order, it is important to only commit tasks when all of their speculation has been resolved. To that effect, the ON checks when all of a task $a$'s predecessors in the task graph have committed, and then it informs the corresponding CN that $a$ is committable. A committable task might not actually commit right away (see Section 3.3). When a subgraph, all of whose predecessors have committed, lies entirely on one CN, that CN can make local decisions about commit order without communicating with the ON.

Each CN executes transactions, caches object values, and prefetches objects to make speculation more effective. SPECTRE makes use of concurrency both within a multi-core CN and across CNs. A single CN runs transactions concurrently on multiple cores, using transactional memory to manage conflicts. Each CN tracks the subset of the global task graph that has been scheduled locally, and detects local conflicts between reads and writes.

## 3.1 Task execution

Each CN concurrently executes as many tasks as there are cores on its computer, giving preference to transactions that are least likely to abort using topological order on the task graph. Where possible, SPECTRE avoids blocking computations while awaiting communication results.

A transaction can be in one of four states. It starts out *Running*, is *Completed* when all its code is executed, and ends up either *Committed* or *Aborted* depending on the outcome of the commit protocol. The CN tracks reads and writes to shared objects using a per-transaction *ReadSet* and *WriteSet*. A *ForkSet* tracks any tasks a transaction has created. Writes become visible to other tasks on the same CN when a transaction completes, and become visible globally when the transaction commits.

## 3.2 Object caching

Each CN caches multiple versions of objects, and speculatively uses cached values to satisfy a transaction's reads. Stale reads are detected by SNs at commit time, and a prefetching and best-effort invalidation protocol helps to keep cached objects up to date to avoid excessive aborts.

The task graph is used to select which version of an object the CN will use for transaction in a read request. This allows tasks to complete out of order: if $x < z$ and $z < y$ then $z$ will read $x$'s values even after $y$ completes.

## 3.3 Aborting and Committing

If a transaction $x$ aborts, the CN also aborts all transactions that read values written by $x$ and schedules new instances of the aborted tasks. These actions can be decided locally: the ON only needs to be involved if $x$ created new tasks that may have been dispatched to other CNs, in which case those tasks also need to be aborted.

If transaction $x$ writes some value and then $y$ reads it or overwrites it, this creates an ordering constraint: $x$ must appear before $y$ in the serialized order. The CN keeps track of these constraints as a dependency graph.

When a CN $C$ hears from the ON that some transaction $x$ is committable, it must wait until all of $x$'s predecessors in the dependency graph have committed before it can attempt to commit $x$. Commit is implemented using a standard two-phase distributed protocol that includes all the SNs that store any object in the union of $x$'s *ReadSet* and *WriteSet*.

## 3.4 Allowing side effects

In practice, a program must produce output beyond merely updating its own set of shared objects. We provide extensions to allow side effects that are familiar from other transactional memory systems. Specifically, SPECTRE supports commit and abort actions [11, 26]: When a transaction commits or aborts, its `OnCommit` or `OnAbort` method is called, providing the transaction with a mechanism to make side-effects visible or clean up after side-effects which must be reversed.

# 4. EVALUATION

While our system implementation is still a work in progress, we have implemented a full distributed prototype of SPECTRE and include one microbenchmark to motivate our hypothesis that transgressive speculation can lead to increased performance. Figure 4 shows timings for a benchmark that computes 800 elements of the series $x_i = x_{\lfloor i/8 \rfloor} \times x_{\lfloor i/9 \rfloor}$, where $x_0$ is a random $200 \times 200$ matrix. The results show that the overhead of our speculation is about 18%, and on this workload transgression is beneficial from two cores onwards. While it is premature to generalize this to real-world workloads and larger systems, we believe it demonstrates the promise of the approach.

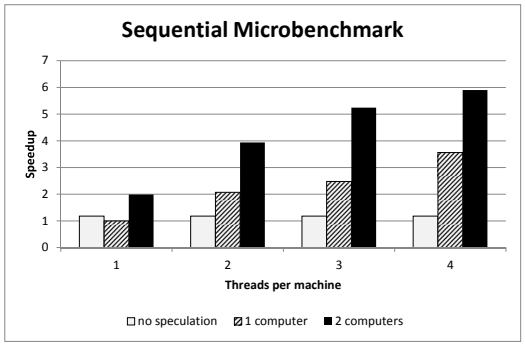


**Figure 4: Speculation finds parallelism. Speedup is relative to the 1-node speculative case.**

# 5. WORK IN PROGRESS

This section describes a number of features that we are planning to implement as we continue to develop the SPECTRE system. They are all motivated by our overall goal of supporting realistic workloads with high performance.

## 5.1 Advanced contention management

Currently, when we abort a completed transaction $x$ we also abort any transactions that speculatively read values written by $x$. There are cases, however, where this is unnecessarily conservative: consider for example a transaction $x$ that removes the head of a priority queue and passes it to some other transaction $y$ that runs an expensive computation. $x$ may have run speculatively out of order with another transaction $z$ that modifies the priority queue, and so $x$ must be aborted and re-run when $z$ completes. If, on re-running $x$, the same object is at the head of the priority queue as before, there is no need to abort $y$ and by tracking object values as well as their versions we can implement a contention-management policy that avoids the cost of re-execution in such cases.

## 5.2 Scheduling and locality

Co-locating computations with the SN that hold their data may be crucial for the performance of some workloads. We plan to investigate scheduling policies that leverage annotations placed on tasks describing the objects they are likely to read. There is a tradeoff between balancing load effectively and ensuring data locality, and it may turn out to be beneficial to allow object migration between SNs to improve overall performance. Annotations that predict access patterns can also be used by a CN to prefetch objects not already present in its cache, either before or during task execution.

## 5.3 Decentralized task graph

We anticipate that the centralized ordering node will become a bottleneck in our system. Although speculative computation can proceed on any CN while it is waiting for the ON to catch up with delayed commit notifications, deep speculation past commit entails overheads in the form of larger object version data-structures, and higher likelihoods of cross-CN conflicts yielding a higher abort rate. It is therefore natural to consider a decentralized implementation for the task graph in which the CNs communicate directly with each other about which tasks have committed and which are committable. With a decentralized graph, we may continue to use a centralized scheduler, or move to a work-stealing design like that used in the Cilk [1] system.

## 5.4 Conflict detection strategy

Our current design detects local conflicts at a CN when a transaction completes, and global conflicts when it attempts to commit. We plan to investigate alternative policies [17, 24, 25], to see which is best suited to our workloads and the distributed nature of the SPECTRE system. For example, an eager distributed conflict detection scheme allowing SNs to broadcast object writes to CNs may benefit some workloads by eliminating work wasted executing doomed transactions. Broadcasting all writes may prove too expensive, so annotations on objects or on writes may help the system prioritize update messages.

# 6. RELATED WORK

SPECTRE combines techniques from transactional memory and distributed shared memory. DSM presents a simple interface to mutable data, and STM is used to enable speculation in order to hide the latency resulting from synchronization and accessing remote data.

With relation to the other TM work, SPECTRE is an object-granularity [10], multi-versioned [3, 22] distributed STM system. SPECTRE has visible readers [23] and relies on eager conflict detection [17] at local nodes, while readers are invisible across CNs, and a lazy conflict detection [24] scheme is used to detect cross-machine conflicts based on commit-time validation of read-sets [6]. SPECTRE implements dependence-awareness [20, 21], allowing concurrent transactions on a single machine to share uncommitted state. A comprehensive review of the TM literature through 2010 can be found in [8].

Our emphasis on speculation guided our design. All-transactional programs have been proposed before [15], to simplify programming, but to the best of our knowledge, SPECTRE is the only TM design that allows the system to speculate past control flow boundaries between successive transactions. SPECTRE goes against received wisdom because it is not opaque [7]. Opacity requires that all transactions (not just the ones that will commit) see a consistent view of the memory. However, we do not believe this is a problem for SPECTRE because it executes all code transactionally, so the effects of errant transactions are never visible to non-transactional code. Conventional C# sandboxing is used e.g. to prevent native method calls from being based on inconsistent state, and doomed transactions are eventually discovered and aborted by our contention-management

system.

Mechanisms similar to those outlined in Section 5.1 have been developed in the past, for slightly different purposes: Galois classes [14] and transactional boosting [13] allow the programmer to provide inverse operations for concurrent data structures, in order to prevent false conflicts on operations that commute even though they may access some memory locations in common. Abstract nested transactions [9] do the same by isolating a computation. If the outcome is the same (despite false conflicts), then the rest of the transaction need not be aborted.

SPECTRE represents the program as a directed acyclic graph, like Cilk [1] which showed that distributed functional programs can be made to scale. Other systems such as TxCache [19] have explored transactional shared memory. We know of only a few systems that combine a transactional runtime with shared memory: ClusterSTM [2] shows that aggregating communication yields excellent scalability. Dash and Demsky's transactional DSM [4, 5] shows that prefetching and caching help mitigate the latency of distributed shared memory, and relies on transactional abort to recover from mis-speculation, as SPECTRE does. However they are not transgressive and wait to know that a transaction has committed before executing beyond it.

# 7. CONCLUSIONS

The SPECTRE implementation is still a work in progress. Our preliminary results convince us, however, that for inherently parallel workloads, transgressive speculation is a powerful tool to hide latencies generated by synchronization and approach an ideal parallel speedup.

# 8. REFERENCES


[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[2] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, 2008.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In *OOPSLA*, 2005.

[4] A. Dash and B. Demsky. Software transactional distributed shared memory. In *PPoPP*, 2009.

[5] A. Dash and B. Demsky. Automatically generating symbolic prefetches for distributed transactional memories. In *ACM/IFIP/USENIX International Middleware Conference*, 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006. Springer-Verlag LNCS Volume 4167.

[7] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, 2008.

[8] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[9] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT*, 2007.

[10] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, Jul 2003.

[11] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS*, 2009.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[13] E. Koskinen and M. Herlihy. Concurrent non-commutative boosted transactions. In *TRANSACT*, 2009.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[15] B. C. Kuszmaul and C. E. Leiserson. Transactions Everywhere, 2003.

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. *CoRR*, 2010.

[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*. 2006.

[18] MPI. http://www.mcs.anl.gov/mpi/.

[19] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, 2010.

[20] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, 2008.

[21] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *PPoPP*, 2009.

[22] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*. Springer, 2006.

[23] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[24] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA*. 2008.

[25] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *MICRO*, 2009.

[26] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT*. 2006.

# A better way to negotiate for testbed resources

Qin Yin, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zürich
{qyin, troscoe}@inf.ethz.ch

## ABSTRACT

Resource allocation is an increasing challenge for distributed network testbeds as computational and network resources are involved. Testbed designers have moved to a query-based model: clients provide a declarative description of their desired resources, and the provider allocate specific resources to meet the request. In this paper, we describe an new approach to negotiate testbed resources between clients and testbed providers: the clients specify their requests as constraints, and the providers reply with resource allocations expressed also as declarative set of constraints on resources. This gives providers more flexibility in late-binding of resources to requests, and opens up a wide design space to optimize resource allocation for efficiency, cost, utilization, or other metrics. Our simple first experiments suggest that the late-binding of resources enabled by representing resource reservation as constraints achieves better network resource utilization compared to the fixed assignment solution.

## 1. INTRODUCTION

Networking testbeds like GENI [7] face an increasing challenge in resource allocation. As dependencies between resources (switch ports, links, virtual machines) become more constrained, resources become more diverse (specialized switches, programmable middleboxes, etc.) and testbeds scale to large numbers of clients, sites, and network elements, it becomes harder for clients to express their requests to the providers of the testbed infrastructure, and in turn for these providers to allocate resources in a way that makes efficient use of the platform.

Faced with these trends designers of testbed platforms have moved to a query-based model for resource request: clients supply a declarative description of resources they want, and the provider allocates (if possible) specific resources to satisfy this request. The request is a set of constraints on the resources to be allocated, and (optionally) some objective function to allow the resource provider to select the "best" (for the client) allocation from the available options.

In this paper, we take the idea a step further: not only do clients specify their requests as constraints, but providers reply with resource promises which are *also* expressed as sets of declarative constraints on resources. We conjecture that this allows providers much greater flexibility in late-binding of resources to requests, and opens up a wide space of techniques for optimizing testbed utilization, including statistical overbooking.

Similar challenges are faced by Infrastructure-as-a-Service (IaaS) providers and scientific Grid platforms, to a lesser, though increasing, degree. We focus on network research platforms in this paper, but we expect that as network topology becomes increasingly important for performance and as a market differentiator [10] in these areas, our ideas will have wider applicability.

In the next section we describe and motivate the specific resource allocation problem we are addressing in more detail, and discuss the background – in particular, why it has not been a problem until recently, and why we think it will become more important in the future.

In section 3 we introduce the idea of expressing resource *advertisements* and *allocations*, as well as simply requests, as declarative sets of constraints. We show this changes the resource negotiation process between clients and providers, and in turn opens up a rich design space for providers to optimize resource allocation for efficiency, cost, utilization, revenue, or other metrics.

We then describe the current status of our work to establish the effectiveness of the idea and show some preliminary results in Section 4.

## 2. BACKGROUND AND MOTIVATION

The basic problem we address in this paper is how a client of an infrastructure provider (e.g. a networking testbed, cloud hosting service, or grid installation) requests resources, how the provider allocates such resources, and how the allocation of such resources is returned to the client. "Resources" in this sense include virtual machines, virtual switches and routers, shares of physical network links, and the like.

At a high level, this process is quite straightforward, and existing testbeds employ simple mechanisms. In this section, we survey how it is done today, and in doing so make the argument that existing solutions to the problem will not suffice for large-scale, distributed facilities where networking resources are explicitly allocated.

### 2.1 Virtual machines

We start with systems that purely allocate virtual machines. Amazon EC2 [5] advertises a small (6 at time of writing) set of VM types (based on location and approximate computing power), each of which consists of a large homogeneous pool. While clients can request VMs in a specific location, EC2's scale means that these


*APSys'11,* July 11–12, 2011, Shanghai, China.

VMs can be allocated entirely independently of each other. Access to the VMs is granted at the time when a request is made – there is no notion of (or need for) future reservations of resources. These factors allow a simple and intuitive API to EC2 and simplify Amazon's task of provisioning physical plant.

PlanetLab [16] is different: clients specify precisely the physical machine on which to create a VM, and all individual physical resources (close to 1000 servers) are visible to clients. Nevertheless, acquiring resources on PlanetLab is conceptually similar to the EC2 case: a request is expressed in terms of the advertised set of machines, and access to resulting "slivers" is granted in the reply. As a best-effort, community-run service, PlanetLab does not need to provide any resource guarantees or make provisioning decisions. Despite this, the diversity of resources offered by PlanetLab (albeit all resembling Linux virtual machines) has led to third-party resource managers [15, 20] which allow clients to specify requests for resources in the form of declarative queries over the available nodes.

This mirrors constraint-matching in Grid computing systems: Condor [4] allocates machines to jobs based on a match-making mechanism called ClassAds. Machine characteristics and job requirements are represented in a common framework making it possible to decide, for a particular request-resource pair, whether requirements are satisfied. RedLine [11] uses constraints to describe resource offerings and requests, interprets resource matching problem as a constraint satisfaction problem and explores constraint-solving technologies to implement matching operations.

Grid systems require more complex resource allocation schemes for two reasons: firstly, they typically allocate a large number of machines or VMs *in a single operation* for parallel compute jobs, rather than the piecemeal sliver allocation which suffices for deploying overlays on PlanetLab. Secondly, the resource requirements of Grid jobs motivate *advance reservation* of a block of machines. For example, the Haizea [19] lease manager is an OpenNebula [14] scheduling module which leases VM resources under a variety of terms, including reservations and queuing of best effort requests. The key observation is that the *dependencies between resources* motivate a richer specification for resource requests.

## 2.2 Network virtualization

Such inter-dependencies become more significant when network, as well as computational, resources are involved. Topological considerations tightly constrain resources: virtual machines and virtual switches must be connected by shares of physical links, for example. This introduces two challenges: firstly, how to express requests for combinations of network and compute resources, and secondly, how to efficiently allocate such resources in the provider.

Emulab [6] solves the former by using `ns2` configurations to express virtual networks, which are then embedded into its physical topology using simulated annealing graph mapping algorithms. It successfully avoids the latter challenge through its focus on centrally-controlled network emulation: Emulab exploits high-capacity network switches which approximate a physical crossbar between machines, thereby rendering the problem of embedding clients' virtual networks in Emulab's physical infrastructure tractable.

However, recent proposals for distributed network testbeds such as GENI will not be amenable to such solutions, since they presume a federated, distributed physical infrastructure over which virtual networks ("slices") are instantiated. The expected low cross-sectional bandwidth in GENI-like testbeds leads to inefficient allocation with simple greedy approaches.

A simple example (Figure 1) should make this clear. Given a physical network of 2 switches and 8 hosts, a simple strategy might allocate for `REQUEST1` 4 hosts under one switch and leave `REQUEST2` unsatisfiable. In fact, both requests can be met with the allocation solution shown on the right.

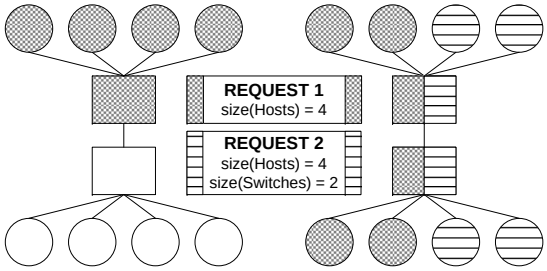


**Figure 1.** Virtual network allocation example

ProtoGENI [17] uses an optimistic reservation mechanism: the users make the resource availability query right before running the slice embedder, then attempt to get tickets for the components that were chosen.

The ORCA-BEN project [3] uses an ontology language called NDL-OWL to express rich resource requests as queries, including topology embedding. Like some other platforms, ORCA differentiates *tickets* from *leases*. The application requests *u* slivers of resource type *r*. If approved, a ticket is issued for *u* units from a specific pool, and the client later redeems the ticket to obtain a lease for the resources.

Such concepts are not totally new: open signaling [2] and systems like Tempest [13] and the Genesis Kernel [8] explored how to virtualize networks efficiently to support multiple, custom control planes over a single physical infrastructure. More recently, OpenFlow [12] has applied similar ideas to IP- and Ethernet-based networks. Based on this, systems like FlowVisor [18] act as transparent proxies between OpenFlow switches and controllers to provide slices of a physical network. In Internet architecture research, so-called "pluralists" even view support for co-existing virtual networks as the key feature of the architecture [1].

Recently, problem has grown beyond academic interest. Topology-Aware Resource Allocation [10] in IaaS-based cloud systems aims to better support data-intensive workloads by making providers more aware of hosted application requirements and giving users fine-grained visibility into, or control over, the infrastructure.

# 3. A (POSSIBLY) BETTER WAY

We are building a resource allocator for testbeds to investigate a different approach: we use constraints to describe not only resource requests, but also resource promises made by the provider.

A request is a list of constraints on:

- when the request should be satisfied (start, end time)
- types of computational and networking resources: characteristics of these resources (CPU, memory of the compute units, table entries of the switches, latency or bandwidth of the links) as well as aggregated properties of the resource set;
- connectivity: topological properties of the virtual network, maximum fanout, network diameter, etc.

A very simple example request may look like this:

```
Time :: 8..12,       % request from 8am for 4 hours
size(Hosts) = 3      % three hosts
sum(Hosts, cpu) > 8 % total CPU units larger than 8
size(Switches) >1    % more than one switch
di(Topology) = 3     % network diameter is 3
```

We retain the distinction between tickets and leases: a lease grants access to specific, named resource components, whereas a ticket simply describes (in more or less detail) a set of resources which the client may gain a lease to in the future.

In previous systems, a ticket, signed by the provider, serves as a reservation of a set of concrete resources – a promise by the provider to bind the resources to the client and grant access to them in the future.

In contrast with those systems, the ticket need not bind a specific set of resources, and indeed need not correspond at all with the request that generated it. Instead, the ticket is simply another set of constraints. These might refer to specific switches or nodes, but are more likely to be "unbound": they simply describe elements in the virtual network are not yet mapped to a physical component.

Only leases will always refer to specific resources. Exchanging a ticket for a lease grants access to specific concrete components over a definite interval of time.

## 3.1 Client-Provider interaction

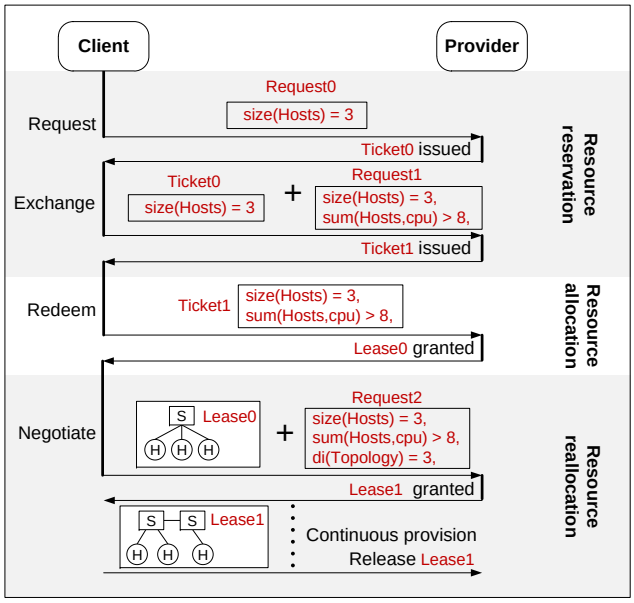


**Figure 2.** Resource negotiation

Figure 2 depicts how client and provider negotiate testbed resources. This is similar to the protocols used in other systems, but the significant differences are, firstly, that tickets generally do not specify definite resources (or even definite numbers in some cases), and secondly, that resource negotiation is ongoing: a client can at any time return a ticket (or a lease) and exchange it atomically for some other ticket or lease.

In the example, the client sends $Request_0$ for 3 hosts. The provider accepts this request and issues $Ticket_0$ for 3 (non-specific) hosts. Later, the client refines its request, returns $Ticket_0$ and requests in exchange 3 hosts with at least 8 CPU units. This request is again satisfied and $Ticket_1$ returned. In this resource reservation phrase, tickets are "unbound", and the client is not subject to failures of specific nodes. Therefor, the client does not have to frequently check the availability of the reserved "unbound" nodes.

Eventually, the client presents the ticket and requests a lease for the resources. Only then, the provider will allocate specific resources and grant the client a lease after which the client gets the control over the resources.

Even after resource allocation (the ticket is redeemed), the client is still able to negotiate with the provider by sending a new refined request and returning the granted lease. In the example, the client adds an additional constraint on network diameter. For the provider, to minimize the overhead of resource reallocation, it's preferrable to keep as much current allocation possible (two hosts under one switch), release the last host under this switch, and alloate another switch with one host connected to it.

More general operations such as splitting and merging of tickets are also possible, in line with existing testbed proposals.

Of more interest, however, than the protocol itself is the state maintained by both sides and the actions taken on receipt of a message, in particular, in the provider.

The provider maintains an up-to-date list of all physical resources available, and their current condition, together with a list of all valid tickets and leases that it has issued. A provider will typically also have some kind of objective function it will seek to maximize – mostly likely utilization in the case of a networking testbed, but in commercial settings this will probably be some function of yield or revenue.

Conceptually, when it receives a request for a ticket, the provider will try to generate (or, strictly speaking, *prove the existence of*) an assignment of physical resources to a new ticket which optimizes its object function subject to the set of constraints imposed by:

1. the availability and topology of physical resources
2. the set of already-issued tickets and leases, *minus* those being returned in the request

If the request is for a lease, the provider will return this concrete assignment in the lease. This operation may, of course, fail: changes in testbed since the ticket was issued may result in the provider being unable to satisfy the request. However, if the request is for a ticket, the provider has considerable freedom in deciding whether to issue a ticket, and for what.

The straightforward procedure above which takes all allocations into account can give optimal use of the infrastructure (without resorting to revoking leases), and therefore extracts maximum benefit from the fact that tickets do not imply definite assignments. Solving this problem includes embedding many virtual networks into a physical network, and is known to be NP-hard. However, there may be approaches which produce near-optimal solutions cheaply.

Another approach is to assign resources as we go: the provider retains the previous concrete assignment and assigns resources for a new request only from previous unassigned ones. This incremental approach is the behavior of current testbeds.

Between these extremes there is a trade-off: the more existing reservations our system can reconsider, the greater the complexity of allocation but the higher the potential efficiency of the result. We show preliminary investigations of this trade-off in the next section.

Our point is not to identify an optimal algorithm at this stage, but rather to show that there is a wide space of possible solutions. Some may be appropriate for academic testbeds which follow PlanetLab's community model, while others may employ sophisticated yield-management techniques (as in, for example, the travel industry), including the use of overbooking and variable price models, to maximise commercial revenue.

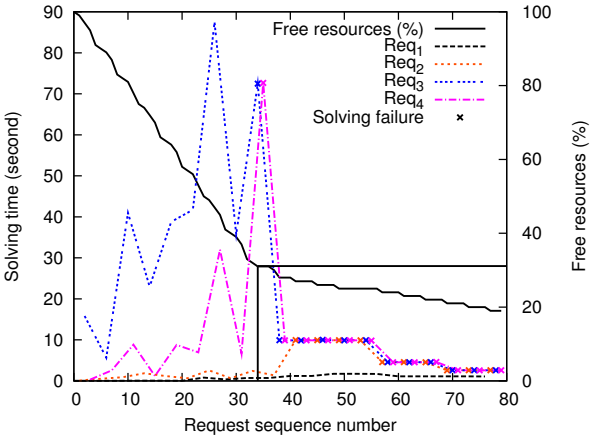


**Figure 3.** Sequential solving

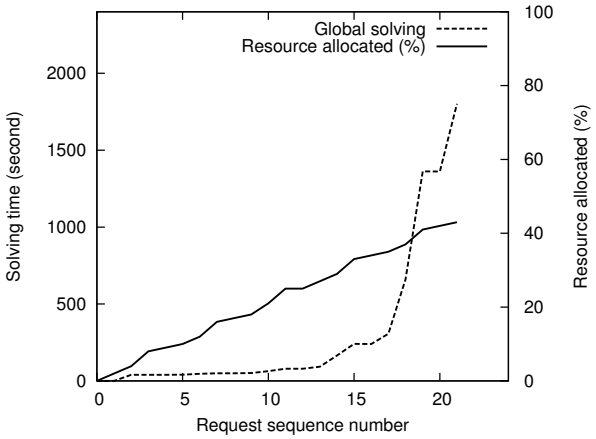


**Figure 4.** Global solving

## 3.2 Further advantages

As well as opening up a wide space of allocation strategies, issuing tickets as constraints offers many other potential advantages for network testbeds. Firstly, late-binding of resources accommodates changes in physical resource availability. Failures can be masked if resources can be reallocated before they are required.

Secondly, resources can be specified with different levels of generality. Very general requests such as "4 machines for 2 hours in the morning" are both easy to express and fit naturally into the provider's framework. Furthermore, in the case of network elements with varying capabilities, we avoid committing a machine early to a task which does not fully exercise it. Finally, multi-stage negotiation is possible with constraint-based tickets. The client is able to refine the resource request by adding or modifying constraints.

# 4. INITIAL RESULTS

We now briefly present some initial results aimed at establishing the feasibility of our approach. In particular, we are interested in the *size of space* for optimization that is opened up by late-binding resource requests. A thorough, real-world evaluation is beyond the scope of this paper and a topic of our ongoing work.

We used Mininet [9] to generate a physical tree network (depth 3, fanout 6) with 216 hosts and 43 switches, and randomly annotated the nodes with different capabilities. Our test workload is a round-robin sequence of 4 pre-defined requests, $Req_1$ and $Req_2$ are simple requests for networks of 2 and 5 nodes respectively, while $Req_3$ and $Req_4$ are more complex requests for larger networks (7 and 11 nodes) with specific topologies. For different allocation strategies, we are interested both in time to perform successive allocations, and the total proportion of physical resources that can be allocated.

In our first experiment we allocate resources to each request when it arrives, and never reallocates resources, roughly following the behavior of current systems. As Figure 3 shows, after the first 34 requests are satisfied (70% utilization), only small requests can be met. Note also that solving time (whether successful or not) decreases as available resources are reduced.

Second, we try full resource reallocation: as each request arrives, we globally allocate all resources to the complete set of requests so far with no a priori allocations. This problem is NP-complete, and so as Figure 4 shows, execution time increases exponentially and after several hours, our solver fails to allocate even half of the available network.

Finally, we pick a design point between these two: allocate sequentially as in the first experiment case, but when allocation fails retry by remapping, in one go, the current request and all existing requests. These requests are ordered by their complexity, and solved independently as we did in the first experiment. Intuitively, this represents a compromise between the exponential solving time of reconsidering all prior requests altogether, and the severely constrained approach of fixing previous allocations. Remapping the more complex requests first might be expected to result in more freedom to find space for new requests.

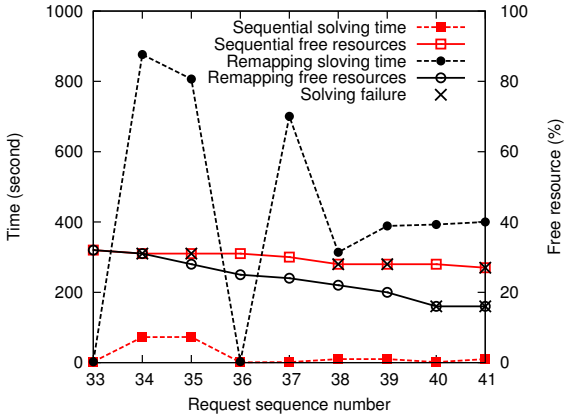


**Figure 5.** Constraint remapping

Figure 5 shows the results. Versus the simple sequential approach, runtime is much higher (though not prohibitive, even with our unoptimized solver). We satisfy more requests (we now fail first at request 40), resulting in greater utilization (16.7% remaining free).

# 5. DISCUSSION

There is much scope for improving the run time of our solver for remapping resources (better algorithms such as simulated annealing, a more sophisticated solving engine, etc.). We conjecture that there are also better heuristics for resource remapping than the sim-

ple one we evaluate here: the field of Operations Research has a wealth of results on this kind of problem.

We are also only beginning to explore the possibilities for overbooking and online resource renegotiation permitted by this approach. However, even our simple first experiments suggest that the late-binding of resources enabled by representing resource tickets as constraints opens up a significant space for optimization of resource usage by platform providers.

## References


[1] ANDERSON, T., PETERSON, L., SHENKER, S., AND TURNER, J. Overcoming the internet impasse through virtualization. *Computer 38* (April 2005), 34–41.

[2] CAMPBELL, A. T., DE MEER, H. G., KOUNAVIS, M. E., MIKI, K., VICENTE, J. B., AND VILLELA, D. A survey of programmable networks. *SIGCOMM Comput. Commun. Rev. 29* (April 1999), 7–23.

[3] CHASE, J. ORCA control framework architecture and internals. Technical report, Duke University, September 2009.

[4] Condor high throughput computing. http://www.cs.wisc.edu/condor/.

[5] Amazon elastic compute cloud (amazon EC2). http://aws.amazon.com/ec2/.

[6] Emulab - Network Emulation Testbed. http://www.emulab.net/.

[7] Global environment for network innovations (GENI). http://www.geni.net/.

[8] KOUNAVIS, M. E., CAMPBELL, A. T., CHOU, S., MODOUX, F., VICENTE, J., AND ZHUANG, H. The genesis kernel: A programming system for spawning network architectures. *IEEE Journal on Selected Areas in Communications 19* (2001), 511–526.

[9] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets '10, ACM, pp. 19:1–19:6.

[10] LEE, G., TOLIA, N., RANGANATHAN, P., AND KATZ, R. H. Topology-aware resource allocation for data-intensive workloads. *SIGCOMM Comput. Commun. Rev. 41* (2011), 120–124.

[11] LIU, C., AND FOSTER, I. A constraint language approach to matchmaking. In *Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)* (Washington, DC, USA, 2004), RIDE '04, IEEE Computer Society, pp. 7–14.

[12] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38* (March 2008), 69–74.

[13] MERWE, J. E. V. D., ROONEY, S., LESLIE, I. M., AND CROSBY, S. A. The tempest - a practical framework for network programmability. *IEEE Network 12* (1997), 20–28.

[14] OpenNebula: The Open Source Toolkit for Cloud Computing. http://opennebula.org/.

[15] OPPENHEIMER, D., ALBRECHT, J., PATTERSON, D., AND VAHDAT, A. Distributed resource discovery on PlanetLab with SWORD. In *WORLDS'04* (Dec. 2004).

[16] PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.

[17] ProtoGENI. http://www.protogeni.net/trac/protogeni.

[18] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[19] SOTOMAYOR, B., MONTERO, R. S., LLORENTE, I. M., AND FOSTER, I. Resource leasing and the art of suspending virtual machines. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 59–68.

[20] YIN, Q., SCHÜPBACH, A., CAPPOS, J., BAUMANN, A., AND ROSCOE, T. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2009), Middleware '09, Springer-Verlag New York, Inc., pp. 10:1–10:20.

# Attendees

Sidney Amani (NICTA / UNSW)
Yungang Bao (Princeton University / ICT)
Bernard Blackham (NICTA / UNSW)
Tim Brecht (University of Waterloo)
Gang Chen (Huazhong University of Science and Technology)
Haibo Chen (Fudan University)
Haogang Chen (MIT)
Jin Chen (Fudan University)
Rishan Chen (Peking University)
Rong Chen (Fudan University)
Wenguang Chen (Tsinghua University)
Xiangqun Chen (Peking University)
Yufei Chen (Fudan University)
Xiao Cheng (Shanghai Jiao Tong University)
Baozeng Ding (ISCAS)
Peter Druschel (MPI-SWS)
Zhaohui Du (Intel Labs China)
Xuepeng Fan (Huazhong University of Science and Technology)
Yu Fan (Shanghai Jiao Tong University)
Hanjun Gao (WuHan University)
Ying Gao (Intel Labs China)
Liang Gu (Yale University)
Bei Guan (Institute of Software CAS)
Haibing Guan (Shanghai Jiao Tong University)
Yao Guo (Peking University)
Zhenyu Guo (Microsoft Research Asia)
Jincheng Han (Fudan University)
Wentao Han (Tsinghua University)
Zhijun Hao (Fudan University)
Jiangzhou He (Tsinghua University)
Gernot Heiser (NICTA & UNSW)
Cheol-Ho Hong (Korea University)
Zhenyu Hou (Baidu)
Yabin Hu (Huazhong University of Science and Technology)
Peng Huang (UC San Diego)
Weihang Jiang (Youdao)
Yunyun Jiang (Tsinghua University)
Xinxin Jin (UC San Diego)
Bumsoo Kang (KAIST)
Anil Karna (Shanghai Jiao Tong University)
Junghyun Kim (Seoul National University)

Jaejin Lee (Seoul National University)
Jiping Li (Fudan University)
Kai Li (Princeton University)
Zhenyu Li (Google)
Zongliang Li (EMC)
Zhenkai Liang (National University of Singapo)
Xin Liao (EMC)
Heng Lin (Tsinghua University)
Di Liu (Tsinghua University)
Mingliang Liu (Tsinghua University)
Ran Liu (Fudan University)
Yunxin Liu (Microsoft Research Asia)
Hang Lu (Institute of Computing Technology CAS)
Longwen Lu (Shanghai Jiao Tong University)
Yandong Mao (MIT)
Jean-Philippe Martin (Microsoft Research Silicon Valley)
Sue Moon (KAIST)
Christianto Oeij (JAIST)
KyoungSoo Park (KAIST)
Zhengping Qian (Microsoft Research Asia)
Hao Qin (Huazhong University of Science and Technology)
Christopher Rossbach (Microsoft Research Silicon Valley)
Sangmin Seo (Seoul National University)
Jie Shen (Princeton University)
Jicheng Shi (Fudan University)
Xiang Song (Fudan University)
Qiang Sun (Shanghai Jiao Tong University)
Doug Terry (Microsoft Research Silicon Valley)
Chandu Thekkath (Microsoft Research Silicon Valley)
Animesh Trivedi (IBM Research Zurich)
Qichen Tu (Youdao)
Geoff Voelker (UCSD)
Xi Wang (MIT)
Yuanxuan Wang (Fudan University)
Zhaoguo Wang (Fudan University)
Ming Wu (Microsoft Research Asia)
Yanjun Wu (Institute of Software CAS)
Mingyuan Xia (Shanghai Jiao Tong University)
Yubin Xia (Fudan University)
Feng Xie (Shanghai Jiao Tong University)
Chuan Xu (Baidu)
Ruini Xue (Tsinghua University)
Hiroshi Yamada (Keio University)
Jianian Yan (Tsinghua University)

Yi Yang (Tsinghua University)
Ziye Yang (EMC)
ChenCheng Ye (Huazhong University of Science and Technology)
Qin Yin (Systems Group ETH Zurich)
Chuck Yoo (Korea University)
Seehwan Yoo (Korea University)
Honesty Young (Intel)
Jiageng Yu (Institute of Software CAS)
Miao Yu (Shanghai Jiao Tong University)
Jidong Zhai (Tsinghua University)
Yan Zhai (Tsinghua University)
Cheng Zhang (Shanghai Jiao Tong University)
Di Zhang (Tsinghua University)
Hong Zhang (Tsinghua University)
Lin Zhang (Peking University)
Lintao Zhang (Microsoft Research Asia)
Yu Zhang (University of Science & Technology of China)
Zheng Zhang (Microsoft Research Asia)
Feng Zhao (Microsoft Research Asia)
Jianjun Zhao (Shanghai Jiao Tong University)
Yangyang Zhao (Tsinghua University)
Jingyu Zhou (Shanghai Jiao Tong University)
Lidong Zhou (Microsoft Research Asia)
Yuanyuan Zhou (UCSD)
Yue Zhou (Fudan University)