Increasing the Trustworthiness of Commodity Hardware Through Software

Kevin Elphinstone
NICTA and University of New South Wales
Sydney, Australia
Email: Kevin.Elphinstone@nicta.com.au

Yanyan Shen
NICTA and University of New South Wales
Sydney, Australia
Email: Yanyan.Shen@nicta.com.au

***Abstract*—Advances in formal software verification has produced an operating system that is guaranteed mathematically to be correct and enforce access isolation. Such an operating system could potentially consolidate safety and security critical software on a single device where previously multiple devices were used. One of the barriers to consolidation on commodity hardware is the lack of hardware dependability features. A hardware fault triggered by cosmic rays, alpha particle strikes, etc. potentially invalidates the strong mathematical guarantees.**

This paper discusses improving the trustworthiness of commodity hardware to enable a verified microkernel to be used in some situations previously needing separate computers. We explore leveraging multicore processors to provide redundancy, and report the results of our initial performance investigation.

***Keywords*-multicore; kernel; reliability;**

I. Introduction

High security computer systems that aim to preserve integrity or confidentiality of data still use simple, yet cumbersome techniques, such as an *air gap* to provide strong guarantees of isolation between components of a computer system [1]. This might be isolation of classified information from networks or components of lower clearance, isolation of *red* plain-text data from *black* encrypted data, or isolation of faults from critical infrastructure.

Cloud computing provides many opportunities for more flexible and cost effective computing, however it is difficult to reconcile the cloud's consolidation of computing with the high security community's conservative approach to strong isolation. Approaches such as *multiple independent level of security* (MILS) set a precedent in architecting the consolidation of previously disparate systems [2]. However, there is a lack of trust in software systems to preserve isolation in MILS-like architectures, either due to the potential for bugs in the implementation of a sound design, or due to potential faults in a design itself due to the complexity of the system. The presence of side-channel attacks further reduces the confidence in cloud infrastructure to enforce isolation [3].

Recent advances in verification and microkernel design have progressed to the point of providing mathematical guarantees of functional correctness of the seL4 microkernel [4]. Further research has extended that work to guarantee the integrity of isolated subsystems built upon the seL4 microkernel [5]. Formal assurance of confidentiality (and the practical strength or otherwise of various formulations of confidentiality) is an active area of research. However, we are approaching the point in time where lack of trust in software to preserve acceptable isolation will no longer be the main issue limiting the adoption of MILS-style approaches to security and safety.

For MILS-style systems to succeed with a verified kernel, the assumptions made in establishing the above formal guarantees will need sufficient guarantees that fit the risk profile of the application domain. Some assumptions, like compiler correctness, can be addressed by providing strong mathematical guarantees [6], which is another area of ongoing research. Assumptions, such as hardware correctness, are more difficult to address as even correctly designed and manufactured hardware can experience errors due to cosmic radiation [7], [8].

Our work aims to explore how to increase the trustworthiness of commercial off-the-shelf (COTS) hardware in order to deploy a verified microkernel in application domains which require both high security (or safety) and hardware consolidation. The main focus of this paper is to describe and motivate the problem area, and then examine the performance implications at the microkernel level of using redundancy to improve the trustworthiness of hardware.

II. Background and Problem

Verification of software involves showing that a representation of the software (ideally the machine code itself) adheres to an abstract specification of the software's behaviour. The lowest-level representation of the program defines the assumed behaviour of the machine. If actual hardware behaviour deviates from assumed machine behaviour, any proven properties of software potentially no longer hold.

Fortunately, there is a large body of work in the area of hardware verification. Hardware verification aims to ensure that the processor logic design adheres to the specification of the machine code. Verification involves formal methods combined with traditional simulation [9]–[11]. Post-silicon validation tests aim to ensure the actual products adhere to the machine specifications. For our work, we assume a correct (or at least known) initial hardware design and implementation that forms the foundation of the verified software stack.

However, correctly designed and implemented hardware is not guaranteed to behave correctly indefinitely. Factors including high temperatures, circuit aging, and radiation may still trigger permanent or transient hardware faults [12], [13]. One recent serious service outage of the Amazon S3 system was caused by a singe-bit corruption in several messages [14]. A study of hardware failure rates of one million consumer PCs showed significant failure rates [15].

The flow-on effect of hardware failures on software obviously varies form benign to catastrophic. A recent study showed 1-2% of activated errors injected into the wu-ftpd and sshd resulted in permanent vulnerabilities being opened [16]. A study of security violations introduced by transient errors in the firewall subsystem of the Linux kernel result in 2% of injected errors causing vulnerabilities, with some vulnerabilities requiring rebooting the system to remove [17]. The Java virtual machine bytecode verifier was successfuly attacked via memory soft errors induced with a 50-watt light bulb [18].

There are many approaches aiming to mitigate these hardware faults. We survey various hardware and software approaches later in section V. For now, we make the following assertions.

- Hardware approaches using a combination of redundancy, extensive self-checking circuitry, hardware isolation, or radiation hardening are unlikely to become ubiquitous in COTS hardware in the near future. Such approaches are also at a disadvantage when power consumption is an issue and high-dependability is not, such as in consumer embedded devices.
- Software approaches that replicate computation, at either instruction, process, or virtual machine granularity, assume the correctness of the underlying operating system or virtual machine monitor.

Thus in a general sense, the problem we aim to solve is how to leverage ubiquitous multi-core processors to mitigate hardware faults for a verified operating system, while minimizing the performance impact. More specifically, we aim to:

- maximize the sphere of replication of the trusted software stack, including the operating system kernel itself,
- minimize the performance impact of replication and synchronization of replicas,
- and retain the formal guarantees of the existing formal verification within the replicas.

Of the three points above, this paper is mostly concerned with the second point.

III. Sample Scenario

To facilitate further discussion, Figure 1 shows a sample security architecture where two virtual machines of differing security classifications are co-located on a single machine running the seL4 microkernel. The goal of the architecture is to isolate one security classification from another, i.e. ensure the untrusted Linux virtual machines (VMs) are isolated from each other, while at the same time, minimizing the amount of code trusted to perform correctly. The VMs communicate with the outside world via a cryptographic (de-)multiplexer (e.g. a VPN termination endpoint) that ensures all information is encrypted when passing out of a VM, and decrypted when passing into a VM, thus allowing different classification VMs to share a network connection.

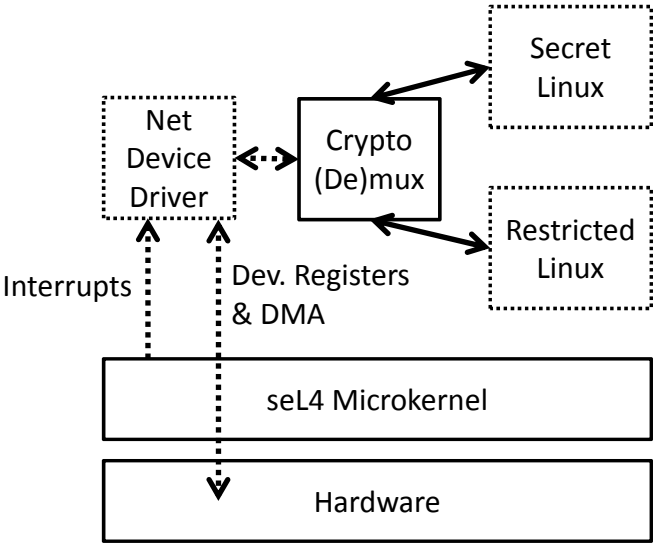

Figure 1. Server consolidation architecture

The seL4 microkernel is responsible for managing the hardware-provided isolation mechanisms to establish the security domains (the 4 upper boxes in the diagram), and to establish only the communication channels indicated by the arrows between those boxes. On seL4, device drivers run at user-level, encapsulated within a security domain just like a VM or traditional process. Where available, seL4 will manage an I/O MMU (such as Intel VT-d) to limit DMA access to within the security domain. Interrupts are delivered via *inter-process communication* (IPC). Thus device drivers have the potential to be untrusted when appropriate hardware is available.

Examining our scenario, given the goal of confining confidentiality of information within the virtual machines, irrespective of their behaviour. We observe that if in preserving confidentiality, we are willing to tolerate denial of service, then the only trusted components in the architecture are the hardware, the seL4 microkernel, and the cryptographic (de-)multiplexer. The driver and two VMs cannot violate the confinement of confidential information unless either:

- the cryptographic component leaks plain text data,
- seL4 fails to enforce the isolation boundaries indicated,
- or the hardware fails to behave correctly.

The seL4 microkernel has been formally verified to be-

have correctly, including being able to control information flow (time-based covert channels are still an open area of research). Similar techniques can be applied to the cryptographic component. Thus the security of the architecture hinges on the correct behaviour of the hardware as we previously introduced.

We now re-examine two of our three aims from section II in the context of our example.

- The maximal sphere of replication is obviously limited by the availability of replicated hardware. Single devices will be a single point of failure. However, multi-core hardware is ubiquitous, which enables the sphere of replication to ideally encompass all trusted software that enforces the security property, if not the entire software stack. The challenge will be minimising the exposure to single point of failure at the single device boundary, in this case the network driver.
  The security property of interest in this case does not rely on trusting the network driver, thus at least for this example, the single point of failure exposes us to denial of service, not violation of our security property (under the assumption that hardware isolation holds in the presence of driver faults). The situation is not as straightforward if the driver needs to be trusted, and thus exposing the trusted software to a single point of failure.
- The performance of microkernel-based systems is critically dependent on the performance of the IPC primitives used to communicate between software components [19]. IPC performance of a microkernel is analogous to the system call performance of a traditional operating system. Any replication needs to ensure IPC performance is not adversely affected.

IV. Initial Experiments

We now present our initial experiments where we evaluate the effect of redundant execution on IPC performance. We evaluate both *dual modular redundancy* and *triple modular redundancy* scenarios on modern multi-core processors using a modified version of seL4. Specifically, we perform our experiments on an ARM Cortex-A9 dual-core processor (a Texas Instuments OMAP4 4460 running at 1.2GHz) on the Pandaboard ES REV B1, with 1GB memory; and an x86-64 quad-core Core i7 870 CPU running at 2.93 GHz with 4GB memory. The hyper-threading and speedstep functions of Core i7 processor are disabled.

We simplify the general problem for this paper by only running IPC microbenchmarks as the application. There are no device drivers nor interrupts. Thus the sphere of replication under evaluation is CPU core, cache, and memory at the hardware level; and seL4 itself and the microbenchmark application.

Our prototype divides the multi-core processors and memory into distinct nodes that execute independently. The seL4 kernel and applications are replicated across the nodes such that each node is a redundantly executing copy of the kernel and application. The modifications to seL4 required to achieve this, including starting in a consistent initial state, and replicating user-level processes, are achieved for now through manual modification of the source code for each replica. We leave the issue of how to systematically create these replicas while preserving the formal verification as future work.

We chose the user-kernel system call boundary as the granularity of comparing execution of the replicas. Each system call must begin with the same inputs from an application and produce the same outputs, i.e. seL4 and applications must behave the same from an application perspective in all replicas. In order to achieve this, the replica nodes must be deterministic in their execution, and not diverge due to variations in relative progress, or scheduling order. Thus any divergence between replicas must be due to an inconsistent behaviour of the hardware across replicas, and thus a fault occurrence. We acknowledge that cross-checking replicas at the system call boundary is insufficient to guarantee the kernel enforces isolation between security domains. We plan to explore a more systematic checking of kernel state related to enforcing isolation in the future. However, our choice of the system call boundary allows us to evaluate what we expect to be the biggest perturbation of IPC performance.

Listing 1 shows a pseudo-code outline of changes to the seL4 system call path to support redundant execution and cross-checking of system call inputs and outputs. The changes save a copy of inputs used by a system call in a region shared between processors, performs the system call, and then copies the system call outputs to the shared area, then waits on a barrier prior to comparing results with a second processor. If the results are consistent, the processor waits for the other processor(s) to complete their check. If a comparison fails, the system is halted.

To evaluate the effect of our change on IPC performance, we measure inter-address space one-way IPC costs for the most commonly used variant of seL4 IPC (a *call*) as our micro-benchmark. Our micro-benchmark uses the CPU cycle counter to time-stamp immediately prior to when a process is *calling* another process, and also when the other process actually receives the IPC. We benchmark two variations. The first benchmark is sending a zero-length message which represents the best-case overhead of IPC with no copying overhead. For cross-checking, a zero-length message still has system call inputs and outputs that describe the message sent and received, which are the inputs and outputs checked via replicas. The *call* is repeated 50 times and the best-case number is reported, so as to estimate the hot-cache performance, and thus the lowest overhead – in practice the overhead will be higher. The second benchmark is the same as the first, except the message size is the maximum size supported by the *fastpath* optimization, which is a carefully crafted code

path aimed at delivering short messages efficiently. In the case of ARM, the messages size is 4 words, for x86-64, 10 words. The results are shown in Table I.

Listing 1. Checking Pseudo Code

```
/* enter the kernel by a system call */
save_input_to_shared_area();

/* call normal kernel path */
handle_system_call();

/* about to leave the kernel */
save_output_to_shared();
/*
 * count is an array can be
 * accessed by all nodes, it is indexed
 * by the node id.
 */
count[my_node_id]++;
paired_node_id = (my_node_id + 1) %
    total_node_number;
/* wait until all counters are equal */
count_barrier();

/* comparing the content with the next
    adjacent node */
correct =
    do_content_checking(paired_node_id);
if (!correct) halt_the_node();

/*
 * increase the counter again and sync
 * again to make sure the checking on all
 * nodes finished successfully
 */
count[my_node_id]++;
count_barrier();

/* now the inputs and outputs are
    consistent */
return_from_syscall();
```

Table I
INTER-AS IPC *Call* BENCHMARK (CYCLES)

Benchmark	Baseline (single core)	DMR	TMR
ARM (0 words)	280	738	N/A
ARM (4 words)	302	1095	N/A
x86-64 (0 words)	784	1336	1560
x86-64 (10 words)	860	1780	2052

DMR: Dual Modular Redundancy TMR: Triple Modular Redundancy

We see that the simple approach to comparing the inputs and the outputs results in a significant overhead being added to IPC. This is attributable to the extra copying of the inputs and outputs to the shared area visible to the other processors, then comparison itself, which results in cache-line transfers between cores, and lastly, the cache-line transfers required when passing the barrier. The overhead obviously increases with size of the inputs and outputs. We expected this overhead to translate into reduced performance at the macro-level, and are actively exploring potential optimisations, in addition to constructing larger systems to quantify the performance reduction.

V. RELATED WORK

A. Hardware

Hardware-based solutions to tolerating intermittent and transient faults have a long history in safety critical computing, such as in aircraft [20], [21]. Using redundant CPUs, memory, buses, and I/O, faults can be detected with dual modular redundancy (DMR), or masked with triple modular redundancy (TMR). Characteristics of solutions in the space are self-checking or redundant vote verification of outputs and configurable isolation of failed components.

High-availability server hardware also uses DMR and TMR in locked-step configurations, where individual microprocessors are synchronised to the same clock and inputs, and the processor outputs are compared to detect or mask faults. More recently, HP NonStop Advance Architecture (NSAA) loosely couples chip multiprocessors to achieve DMR or TMR [22]. Instead of lock-stepping, NSAA deterministically executes the same instruction streams at slightly differing rates, and synchronises and compares I/O operations to redundant devices over redundant system-level communication fabrics. The loose coupling is required as lock-stepping is not possible in the presence of modern processor features such a clock-scaling for power management. The loose coupling also enables one CPU to execute recovery code in the case of a faults, or TLB miss code, and still eventually produce the same I/O output as non-faulting code. They rely on hardware implementing *fail-fast*, and not continuing erroneous execution, nor propagating faults across CPUs. Fail-fast is not guaranteed in the case of commodity CPUs.

Researchers have also observed that commodity chip multiprocessors lack strong fault isolation between cores on the chip, and have proposed configurable isolation between cores [23]. They show graceful degradation over time by simulating reconfiguration in the presence of component failures. This approach provides stronger guarantees against fault propagation, but still relies on "external" result comparison in the I/O subsystem or other external checker not present in commodity platforms to achieve fault masking. Configurable isolation is complementary to our approach, but configuration would require a consensus across more than a single core to strengthen isolation in the presence of an erroneously executing core.

B. Software

Most of the software based solutions focus on application-level fault tolerance or service availability. Compiler-based techniques, such as SWIFT [24] generates binary code with redundant computation and compares the results to detect erroneous computation. This approach does not take the advantage of multi-core processors as the duplicated code runs

on single core only. The comparisons are not self-checked and the approach assumes writes to memory are error-free. Wang et al. [25] introduced compiler-managed redundant multi-threading to utilise the multi-core processors for more efficient transient fault detection. The management layer used to create and schedule the leading/following threads is not protected. PLR (Process-Level Redundancy) [26] applies a sphere of replication to application level, the applications and libraries are replicated to leverage multi-core processors for transient fault tolerance. The PLR management layer and the operating system are assumed correct. EVE (Execute-Verify) [27] applies state machine replication to improve the dependability of the services provided by multi-core servers. Hardware transient faults in the services are detected during the verify stage. Romain [28] is an operating system service which provides transparent redundant multi-threading to tolerate transient faults. The replicated applications states are compared before the states are externalised. The authors note that the Romain service and the kernel are assumed to be reliable and should be protected by other measures [29].

Bressoud et al. implemented a hypervisor-based fault-tolerant system [30] on the HP PA-RISC processors. Protocols were designed to enhance the hypervisor to create and coordinate the primary VM and backup VM to achieve fault tolerance. As the virtualisation technologies are becoming mature, the overhead to create and run virtual machines has reduced significantly. VMWare designed fault-tolerant features for their enterprise product line [31]. The backup VM keeps its internal state synchronised with the primary VM by executing all the events sent by the primary VM through a logging channel. These virtual-machine-based solutions focus on providing high service availability and fail-over is used to mitigate a detected failure. If the hardware faults affect the hypervisor or the hosting operating system, the faults may not be detected and the result could be service outage or data corruption.

VI. Conclusions

We have argued that a formally verified microkernel (such as seL4) provides the high-level correctness and isolation guarantees required to build a trustworthy software system. However, commodity hardware lacks high-dependability features and thus is susceptible to temperature, radiation, cosmic rays and other environmental factors. Transient faults or any other deviation in the assumed hardware behaviour will potentially invalidate the correctness and security guarantees.

In this paper, we have explored leveraging redundant processors to improve the trustworthiness of COTS hardware. We have implemented dual- and triple-redundant versions the seL4 microkernel, and identified inter-process communication as an issue in retaining performance of the microkernel. We have micro-benchmarked both dual- and triple-redundant version of the kernel on both ARM and x86-64 and have observed that there is a significant performance overhead at the micro-benchmark level.

We plan to explore three general areas in the future: (1) evaluating performance at the macro-level of a more significant system, (2) increasing the sphere of replication to check more than just syscall inputs and outputs, and (3) integrating device drivers into our system.

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.

References

[1] National Security Telecommunications and Information Systems Security Committee, National Security Agency, "Red/black installation guidance," Dec. 1995, retrieved from: http://cryptome.org/tempest-2-95.htm (Sept, 2012).

[2] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison, "The MILS architecture for high-assurance embedded systems," *International Journal on Embedded Systems*, vol. 2, pp. 239–247, 2006.

[3] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, USA, 2012, pp. 305–316.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct. 2009, pp. 207–220.

[5] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *2nd International Conference on Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds., vol. 6898. Nijmegen, The Netherlands: Springer, Aug. 2011, pp. 325–340.

[6] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[7] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design and Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[8] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, vol. 25, no. 6, pp. 30–39, Nov. 2005.

[9] B. Bentley, "Validating the Intel Pentium 4 microprocessor," in *Proceedings of the 38th Design Automation Conference (DAC)*, Las Vegas, NV, USA, 2001, pp. 244–248.

[10] J. Bhadra, M. S. Abadir, L. C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Design and Test of Computers*, vol. 24, no. 2, pp. 112–122, Mar. 2007.

[11] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodovä, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," in *Proceedings of the 21st International Conference on Computer Aided Verification*, Grenoble, France, 2009, pp. 414–429.

[12] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer," *IEEE Transactions on Devices and Materials Reliability*, vol. 5, no. 3, pp. 329–335, Sep. 2005.

[13] D. Lyons, "Sun screen," Nov. 2000. [Online]. Available: http://www.forbes.com/global/2000/1113/0323026a.html

[14] Amazon. (2008, Jul.) Amazon s3 availability event: July 20, 2008. [Online]. Available: http://status.aws.amazon.com/s3-20080720.html

[15] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the 6th EuroSys Conference*, Salzburg, Austria, Apr. 2011.

[16] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer, "An experimental study of security vulnerabilities caused by errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2001, pp. 421–430.

[17] S. Chen, J. Xu, Z. Kalbarczyk, R. K. Iyer, and K. Whisnant, "Modeling and evaluating the security threats of transient errors in firewall software," *Performance Evaluation*, vol. 56, no. 1–4, pp. 53–72, Mar. 2004.

[18] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *IEEE Symposium on Security and Privacy*, 2003, pp. 154–165.

[19] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger, "Achieved IPC performance (still the foundation for extensibility)," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, USA, May 1997, pp. 28–31.

[20] A. L. Hopkins Jr., T. B. Smith III, and J. H. Lala, "FTMP—a highly reliable fault-tolerant multiprocess for aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[21] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.

[22] D. Bernick, B. Bruckert, P. Del Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[23] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, USA, 2007, pp. 470–481.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing (Lake George), New York, USA, Oct. 2003.

[25] C. Wang, H. S. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *Proceedings of the 5th International Symposium on Code Generation and Optimization*, 2007, pp. 244–258.

[26] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*, Jun. 2007, pp. 297–306.

[27] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-verify replication for multi-core servers," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, Hollywood, CA, USA, 2012, pp. 237–250.

[28] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the 12th International Conference on Embedded Software*, Tampere, Finland, Oct. 2012, pp. 83–92.

[29] B. Döbel and H. Härtig, "Who watches the watchmen? protecting operating system reliability mechanisms," in *Proceedings of the 8th Workshop on Hot Topics in System Dependability*, Hollywood, CA, USA, Oct. 2012.

[30] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems*, vol. 14, pp. 80–107, 1996.

[31] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM Operating Systems Review*, vol. 44, no. 4, pp. 30–39, Dec. 2010.