Who Watches the Watchmen? – Protecting Operating System Reliability Mechanisms

Björn Döbel
Technische Universität Dresden
Dresden, Germany
doebel@tudos.org

Hermann Härtig
Technische Universität Dresden
Dresden, Germany
haertig@tudos.org

Abstract

We present the design and initial evaluation of a resilient operating system architecture that leverages HW architectures combining few resilient with many non-resilient CPU cores. To this end, we build our system around a *Reliable Computing Base (RCB)* consisting of those software components that must work for reliable operation, and run the RCB on the resilient cores. The remainder of the system runs replicated on unreliable cores. Our system's RCB consists of an L4 microkernel, a runtime environment and a replication manager. In this paper we state and justify assumptions about the hardware architecture, motivate the corresponding software architecture and evaluate communication mechanisms between the RCB and the replicas.

1 Introduction

With every new processor generation used in general purpose computers the number and complexity of functional units increases, which is made possible by decreasing structure sizes. The downside of this development is that processors become more vulnerable to permanent and transient hardware errors [7]. This trend is expected to increase and poses a serious threat to future hardware generations [16].

In this paper, we focus on transient faults caused by *single-event upsets (SEUs)* [18], which are commonly caused by particle strikes, e.g., cosmic radiation. An SEU striking a transistor may result in a corresponding register to change its state, which in turn may lead to erroneous behavior visible at the software level.

Researchers and computer system vendors introduced various mechanisms to cope with SEUs. Hardware-level approaches introduce dedicated hardware circuitry that validates computations [2] and transistor timing [9], enhances data passing through the CPU with redundant signatures [19], or performs lock-stepped execution of processing units [14]. Unfortunately, such approaches are often too expensive (in terms of energy consumption, production cost, or runtime overhead) to be employed in general purpose computer systems.

Software-level solutions exploit application and developer knowledge to decrease the runtime overhead required by a specific fault tolerance strategy [24]. These approaches are often built into a compiler [21], which requires the applications' source code to be available. Applicability of these strategies is limited, because many applications either make use of third-party libraries (often only distributed in binary form) or are provided through mobile app stores where users have no control over the tools used for implementing these applications. Furthermore, most of these solutions are only developed for user-level software [10, 21, 22, 24], whereas the operating system kernel and low-level services are implicitly assumed to work correctly. The few existing whole-system solutions [5, 15] are prohibitively expensive for general purpose computers.

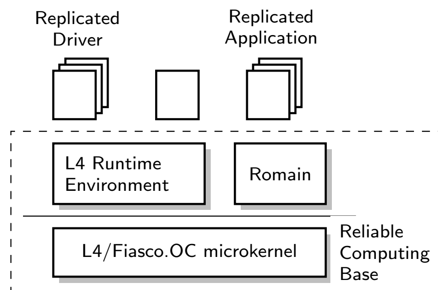

Figure 1: ASTEROID Resilient OS Architecture

In contrast, it is our goal to support reliable operation of binary-only applications, to include operating system services, and to use inexpensive, heterogeneous multicores. We strive to design an operating system architecture that continues correct execution in the presence of transient hardware faults. We focus on SEUs affecting

functional units of the CPU, whereas we assume memory contents to be protected by hardware mechanisms, such as Error-Correcting Codes (ECC) [18].

Our approach is based on low-overhead replicated execution of processes, called *redundant multithreading (RMT)* [20]. In contrast to previous approaches [22, 24], we implement RMT as an operating system service, called Romain. In this paper we focus on the lower-level part of the system: In order to reliably provide RMT, a subset of software components always needs to function correctly. We call this subset the *Reliable Computing Base (RCB)* and want find out how to protect it against SEUs. The overall architecture, called ASTEROID, is depicted in Figure 1.

We briefly describe our implementation of RMT as an OS service in Section 2. In Section 3 we review existing trends in processor architecture and reason that a heterogeneous architecture consisting of few resilient CPU cores and many non-resilient ones appears a promising way of protecting the RCB. In order to combine the ASTEROID design with such a hardware platform we require efficient signalling between CPUs. We evaluate three different signalling implementations in Section 4. Finally, we review the resulting system design and identify points in our system that still remain vulnerable to SEUs in Section 5. We show that the software and hardware components to address these issues already exist, although nobody has yet combined them to provide fault-tolerant execution.

2 Redundant Multithreading as an Operating System Service

We now briefly describe the current state of ASTEROID, our resilient operating system shown in Figure 1. For a more detailed description, please refer to [8]. ASTEROID splits the system into two layers: user applications and the Reliable Computing Base. We detect hardware errors that occur during the execution of user applications by transparently replicating those applications and comparing their states before any state is externalized. Our implementation, Romain, is based on the idea of redundant multithreading (RMT) [20, 22, 24].

A replicated application as depicted in Figure 2 is created by instantiating a master process. The master process then creates identical replicas of the application, each running in a dedicated address space in order to achieve isolation in the case of a faulting replica. Leveraging manycore systems, we schedule each replica on its own physical CPU core so that replicas can execute independently as long as they don't interact with the outside world.

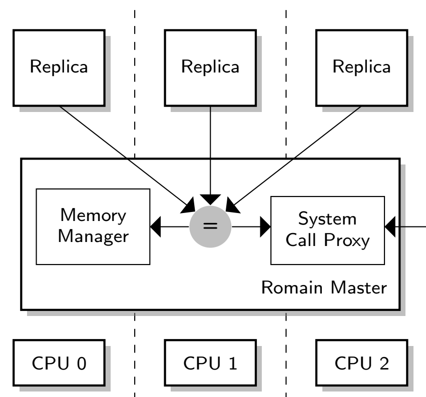

Figure 2: Replicated Application

Whenever a replica thread raises a CPU exception (the most important ones being system calls and page faults), the kernel migrates the thread from the replica to the master address space. The faulting thread then waits for all other replicas to do the same. At this point the replicas have reached an identical state if no hardware fault occurred. After comparing all replica states, one replica handles the fault locally on its own CPU: page faults are resolved directly within the master, system calls are redirected to the underlying kernel. The result of this handling is applied to all other replicas and thereafter each replica moves back to its respective address space and resumes execution. For later reference, we call this fault handling mechanism *Local Fault Handling*.

ASTEROID is based on the L4/Fiasco.OC microkernel and the L4 Runtime Environment (L4Re) [17]. Building on this design, we run traditional operating system services, such as device drivers and protocol stacks as user-space applications. This has the key advantage that most OS services can be protected against SEUs by employing replicated execution.

The RCB of our system comprises the microkernel, a couple of L4Re services and the Romain replication framework. Unfortunately, we cannot protect the RCB against hardware errors using the RMT mechanism we implemented for user applications. Hence, we need to harden it using a different approach, which is the contribution of this paper.

3 Protecting the RCB

The elements of the RCB are under our full control and with their source code available, we could use compiler-based solutions for protecting against hardware errors. Techniques such as software-encoded processing [10] claim to detect all kinds of erroneous deviations. However, these have only been developed and tested for user-level software. In order to apply them to kernel code, we

expect that we'd have to pay special attention to asynchronous activities, such as interrupt handlers. Furthermore, we cannot predict the performance impact these techniques would have on kernel code, which is often highly optimized for heavily used code paths. We explore a different direction for protecting the RCB: We believe that protection can efficiently be achieved by relying on hardware support.

Highly resilient applications in space and avionics usually employ radiation-hardened hardware [3], which is unfortunately too expensive to be available in consumer electronics. In contrast, we propose a heterogeneous manycore hardware platform that is based on already existing hardware components and consists of a few resilient and many non-resilient cores. Resilient cores (*ResCores*) are specially designed to deal with SEUs at the hardware level, whereas non-resilient cores (*NonResCores*) are designed using the cheapest possible implementation that fits the hardware vendors' needs (e.g., in terms of area cost, energy usage etc.).

Splitting hardware into these two resilience classes allows for flexibility and increased utilization: instead of over-provisioning hardware resources in order to always guarantee correct execution, our design allows to dynamically assign cores to different purposes. A NonResCore may be used to execute an application that was hardened using compile-time encoding and does not require any replication. Other NonResCores may be assigned to execute an unprotected application in a replicated manner, requiring a ResCore for coordination.

As OS researchers we don't have the expertise to implement the required hardware changes ourselves. We don't know whether these issues should be addressed by replicating functional units, modifying the hardware production process, or relying on alternative implementations of the same hardware feature. However, we observe trends towards heterogeneous platforms in modern hardware architectures and therefore believe that it is realistic to expect the arrival of a ResCore hardware platform in the mainstream market soon.

Back in 1999, Austin proposed the DIVA architecture [2], which contained 'checker cores' built with a larger structure size than other computational units on the same chip. This design was never used in general purpose hardware, though. However, today we are seeing the widespread introduction of heterogeneous hardware in general purpose systems:

- ARM recently introduced its big.LITTLE architecture [1], which combines complex and less-complex processors on a single chip. The purpose of this split is to save energy by keeping the more complex core switched off whenever possible. Nevertheless, we can also see this split as a first step towards making ResCores and NonResCores available.
- The Cell microarchitecture [12] combines one full-fledged processor with eight vector processing units, which can be used for non-arithmetic processing as well. In Section 5 we will see that this microarchitecture has additional benefits with respect to resilient computing.
- Researchers are investigating the use of general purpose graphics processors (GPGPUs) at various levels of the software stack. While this research focusses on the computational power provided by GPGPUs, these findings may be generalized and apply to fault-tolerant systems as well.

Given these observations, we assume for the rest of this paper that a heterogeneous many-core platform consisting of ResCores and NonResCores will be available. We now look at how to design a resilient OS around such an architecture.

4 From ResCores to Reliability

The fundamental idea for protecting the RCB based on ResCores is simple: all RCB code executes on ResCores, whereas everything else runs on NonResCores. This idea is not new. Split operating system architectures, such as FlexSC [23] and Nix [4], demonstrated that splitting the system into dedicated cores for handling certain classes of work allows to efficiently utilize manycore systems. However, their authors did not focus on resilience against hardware faults.

The Romain replication framework allows for a natural split: replicas execute on NonResCores and CPU exceptions raised by replicas are handled by code running on a ResCore. The interesting question is: How do we efficiently switch between these cores? And, can we deal with faults that occur while such a switch is in progress?

We implemented three alternatives for splitting Romain: migration, split handling with synchronous notifications, and shared-memory polling.

1. *Migration* (Figure 3.1): Instead of handling a CPU exception locally on the faulting core, all replica threads are migrated to a ResCore before comparing their states and selecting a replica to handle the fault. After successful fault handling and applying the resulting state changes, the replicas are migrated back to their NonResCores.

2. *Split handling with synchronous notifications* (Figure 3.2): The migration approach requires all replica threads to be migrated, even though only one actually performs any state comparison and fault handling. As an alternative, we spawn a dedicated handler thread on a resilient core. This thread blocks waiting for notifications from replicas. If a

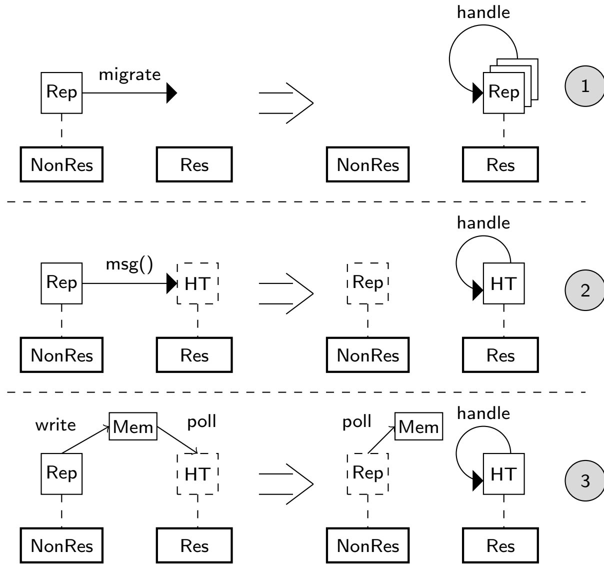

Figure 3: Notification variants: 1) Replica thread (Rep) is migrated from NonResCore to ResCore for fault handling. 2) Rep thread sends a synchronous notification to activate a handler thread on a ResCore. 3) Split handler thread and replica poll on shared memory location for activation.

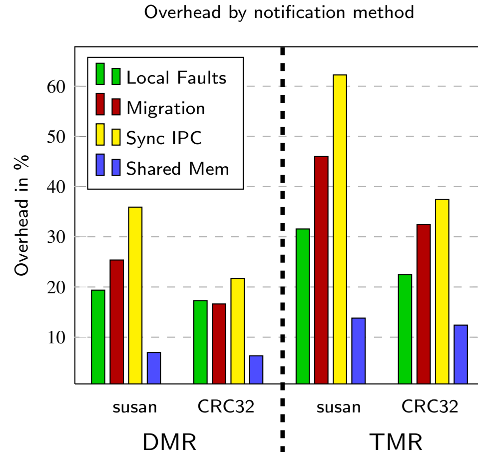

Figure 4: Replication runtime overhead for different notification mechanisms. Overhead is calculated with respect to single-threaded execution in Local Fault Handling mode.

replica raises a CPU exception, it notifies the handler using the synchronous IPC mechanism provided by L4/Fiasco.OC. The replica then blocks waiting for a reply.

The notification wakes up the handler thread, which then waits for notifications from the other replicas. Once these notifications arrived, the handler thread performs state comparison and fault handling, applies results to the replicas, and finally sends wakeup notifications to the replicas.

3. *Split handling with shared-memory polling* (Figure 3.3): The split handling approach above still requires synchronous notifications. In terms of reliability, this means that the underlying kernel mechanism must work correctly at all times. Additionally, FlexSC [23] demonstrated that asynchronous calls through shared memory may lead to better system call throughput and latency than synchronous ones. Therefore, we implemented a third notification mechanism that works similar to the previous one. However, instead of using synchronous IPC, the fault handler thread as well as the replicas await notifications by polling a predefined memory location to switch to a certain value.

We now compare these notification mechanisms to the Local Fault Handling approach described in Section 2. First, all three mechanisms provide fault-tolerant comparison of replica states and handling of CPU exceptions by executing the respective code on a ResCore. In contrast, local fault handling is unaware of the existence of ResCores and hence cannot reliably handle exceptions.

To assess performance, we executed two programs from the MiBench benchmark suite [13] using Romain[1]. We first ran the benchmarks natively on top of the L4/Fiasco.OC microkernel. Thereafter, we executed the benchmarks with Romain in double (DMR) and triple modular redundancy (TMR) using the four different notification mechanisms. The test machine was a computer with 12 physical Intel Core2 CPUs running at 2.6 GHz. Hyperthreading was turned off. Replica threads as well as the RCB exception handler thread were each pinned to a dedicated CPU.

Figure 4 shows the overhead of our benchmarks relative to single-threaded execution. We see that in comparison to local handling (green bar), migrating threads (red bar) or using synchronous notifications (yellow bar) add substantial amounts of overhead. While these effects could be seen as tolerable in DMR, TMR overhead for synchronous IPC notifications is up to twice as high as the overhead for local fault handling.

In contrast, polling a shared memory location (blue bar) appears to be even faster than the local fault handling mechanism. The reason for this observation is that for local fault handling the participating threads need to synchronize and decide who is going to perform the actual handling. In the case of the split handler thread, this is not necessary, as only this dedicated thread is able to

[1] The subset was selected by selecting those benchmarks that showed the highest overheads in [8].

perform exception handling. This result is not surprising, but in line with the observations of Soares' FlexSC [23].

It needs to be noted that these numbers were measured on an SMP architecture providing uniform memory access and cache coherence. Modern multicore systems often use a non-uniform memory hierarchy. Blagodurov et al. showed that performing wrong memory placement in such an architecture may substantially increase memory access latencies [6]. On such systems the benefit of using shared memory polling may decrease.

Experimental architectures, such as the Intel Single Chip Cloud Computer (SCC) [11], remove cache coherence completely. In such an architecture, polling becomes impossible and we need to resort to one of the other notification mechanisms. We draw the conclusion, that our envisioned resilient hardware platform will require a fast inter-CPU messaging mechanism. Such a feature, called Message Passing Buffers, already exists in Intel's SCC.

5 Hardware Requirements

We showed that the ASTEROID resilient system architecture can be mapped to a hardware platform providing ResCores. However, even with such a system in place, there remain points at which an SEU might remain undetected or cause havoc. These points reside at the transitions between non-reliable and reliable execution.

First, our whole system relies on CPU exceptions being triggered at the right point in time. An SEU might lead a CPU to either fail triggering an exception or to trigger a spurious exception. These kinds of errors will be detected by our system: comparison with non-faulty replicas will pinpoint the spurious exception. Using a watchdog mechanism that forces CPUs to trigger an exception after a limited amount of time, failure to do so can be detected as well.

Before notifying the exception handler, the replica thread's state needs to be stored somewhere in memory, so that a separate exception handler thread is able to access it. As a second potential error, the state may become corrupted while writing it to memory. This will also be detected by our architecture once replica states are compared.

Third, any memory accesses performed while executing on a NonResCore may write to the wrong memory location, thereby overwriting either other replicas' states or critical data of the Romain master. As described in Section 2, we use address spaces to isolate replicas. Hence, we require a correctly working MMU on every NonResCore. However, only ResCores must be able to configure the MMU of a NonResCore in order to prevent an SEU from triggering page table entries to be modified. If this is not easily possible, we alternatively imagine an additional layer of memory protection that limits the accessibility of physical memory to certain cores.

Such feature already exists in today's hardware: IOMMUs perform an additional translation between physical and device-physical addresses. Furthermore, Intel's SCC [11] allows configuring memory regions to be accessible only by a specific core. As another solution, in the Cell microarchitecture [12], each core has private memory that no other core can access. Such a design might be extended to implement private memory on NonResCores that can still be accessed by ResCores, so that replica state stored in private memory is safe from faulting replicas, but can still be read and modified by master code.

6 Conclusion

In this paper we presented an overview of the ASTEROID operating system architecture providing transparent redundant multithreading for user-level applications. We pointed out that in order for this system to work, its Reliable Computing Base needs to be protected as well. As a way of protecting the RCB, we proposed the idea of structuring our OS on top of a heterogeneous hardware platform that provides resilient cores to run RCB code and cheap, non-resilient cores to run application code. We evaluated three alternative methods for sending notifications between application and RMT code and discussed the remaining weak spots that may still be vulnerable to SEUs.

Previous approaches tried to address SEUs by inventing completely new hardware. In contrast, we identified hardware features our OS needs to rely on: memory protection using MMUs and CPU-private memory, efficient inter-CPU messaging, and the possibility to implement a watchdog mechanism. These features already exist in state-of-the-art computer architectures. Therefore, it is feasible to expect them to be combined into a completely fault-tolerant architecture.

7 Acknowledgments

This work was partially supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - `spp1500.itec.kit.edu`).

Our colleagues Philip Axer and Michael Roitzsch provided valuable feedback on drafts of this paper. Furthermore, we would like to thank our colleague Michael Engel for fruitful discussions on the concept of the Reliable Computing Base.

References

[1] ARM LTD. Big.LITTLE processing with ARM Cortex-A15. Whitepaper, 2011.

[2] AUSTIN, T. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on* (1999), pp. 196–207.

[3] BAE SYSTEMS. RAD750 family of products. `http://www.baesystems.com/product/BAES_028145`, 2012.

[4] BALLESTEROS, F. J., EVANS, N., FORSYTH, C., GUARDIOLA, G., MCKIE, J., MINNICH, R., AND SORIANO, E. High performance cloud computing is Nix. In *Proceedings of the 2011 Bell Labs Technical Conference* (2011).

[5] BERNICK, D., BRUCKERT, B., VIGNA, P., GARCIA, D., JARDINE, R., KLECKA, J., AND SMULLEN, J. Nonstop: Advanced architecture. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on* (june-1 july 2005), pp. 12–21.

[6] BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the USENIX Annual Technical Conference* (2011), ATC'11, USENIX Association.

[7] BORKAR, S. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro 25*, 6 (Nov.-Dec. 2005), 10–16.

[8] DÖBEL, B., HÄRTIG, H., AND ENGEL, M. Operating system support for redundant multithreading. In *12th International Conference on Embedded Software (EMSOFT)* (Tampere, Finland, October 2012).

[9] ERNST, D., KIM, N. S., DAS, S., PANT, S., RAO, R., PHAM, T., ZIESLER, C., BLAAUW, D., AUSTIN, T., FLAUTNER, K., AND MUDGE, T. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (dec. 2003), pp. 7–18.

[10] FETZER, C., SCHIFFEL, U., AND SÜSSKRAUT, M. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security* (Berlin, Heidelberg, 2009), SAFECOMP '09, Springer-Verlag, pp. 283–296.

[11] GRIES, M., HOFFMANN, U., KONOW, M., AND RIEPEN, M. SCC: A flexible architecture for many-core platform research. *Computing in Science and Engineering 13* (2011), 79–83.

[12] GSCHWIND, M. Chip multiprocessing and the Cell broadband engine. In *Proceedings of the 3rd conference on Computing frontiers* (New York, NY, USA, 2006), CF '06, ACM, pp. 1–8.

[13] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 3–14.

[14] IBM. PowerPC 750GX Lockstep facility. IBM Application Note, 2008.

[15] IBM. z/OS – a smarter operating system for smarter computing. `http://www-03.ibm.com/systems/z/os/zos/`, 2011.

[16] ITRS. International Technology Roadmap for Semiconductors. `http://www.itrs.net/Links/2011ITRS/Home2011.htm`, 2011.

[17] LACKORZYNSKI, A., AND WARG, A. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (Nuremberg, Germany, 2009), ACM, pp. 25–30.

[18] MUKHERJEE, S. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[19] REICK, K., SANDA, P., SWANEY, S., KELLINGTON, J., MACK, M., FLOYD, M., AND HENDERSON, D. Fault-tolerant design of the IBM Power6 Microprocessor. *IEEE Micro 28*, 2 (march-april 2008), 30–38.

[20] REINHARDT, S. K., AND MUKHERJEE, S. S. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News 28* (May 2000), 25–36.

[21] REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization* (2005), IEEE Computer Society, pp. 243–254.

[22] SHYE, A., MOSELEY, T., REDDI, V. J., BLOMSTEDT, J., AND CONNORS, D. A. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2007), DSN '07, IEEE Computer Society, pp. 297–306.

[23] SOARES, L., AND STUMM, M. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[24] WANG, C., KIM, H.-S., WU, Y., AND YING, V. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), CGO '07, IEEE Computer Society, pp. 244–258.