Designing an Analyzable and Resilient Embedded Operating System

Philip Axer, Rolf Ernst
TU Braunschweig
Institute of Computer and Network Engineering
{axer,ernst}@ida.ing.tu-bs.de

Björn Döbel, Hermann Härtig
TU Dresden
Operating Systems Group
{doebel,haertig}@tudos.org

Abstract: Multi-Processor Systems on Chip (MPSoCs) are gaining traction in the embedded systems market. Their use will allow consolidation of a variety of applications into a single computing platform. At the same time the probability of transient hardware malfunctions manifesting at the software level increases.

These applications integrated into the MPSoC possess a range of different requirements in terms of timing, safety and resource demand. Hence, it is impossible to provide a one-size-fits-all solution that allows reliable execution in future MPSoC systems. In this paper, we present ASTEROID, a set of hardware and operating system mechanisms that aim to provide reliable execution in such a diverse environment.

1 Introduction

The self-fulfilling prophecy of Moore's law and the accompanied transistor-shrinking are the major driver of the silicon industry as we know it today. While this development allows for the construction of ever faster processors with larger caches and increasing feature sets leading to multi-core systems and ultimately to many-core architectures, the downside of this development is that chips will be much more prone to temporary or constant malfunction due to a variety of physical effects.

Variability caused by future lithographic process (22nm feature size vs. 193nm wavelength) increases the standard deviation of gate properties such as its delay [Nas10]. The increased likelihood of manufacturing errors may lead to chips with functional units that don't work right from the beginning [Bor05], or to transient effects that only appear under environmental stress. Heat flux and wear-off (e.g. electro-migration) can cause functional units to stop working after some time either for the duration of certain environmental conditions, such as temperature being above a certain threshold, or forever [Sch07]. Overeager undervolting or frequency scaling may lead to another group of errors where signals take too long to reach their destination within the setup and hold-time bounds or where crosstalk between unrelated wires induces voltage changes that become visible at the software level [ZMM04]. Also, radiation originating from space or the packaging of the chip may

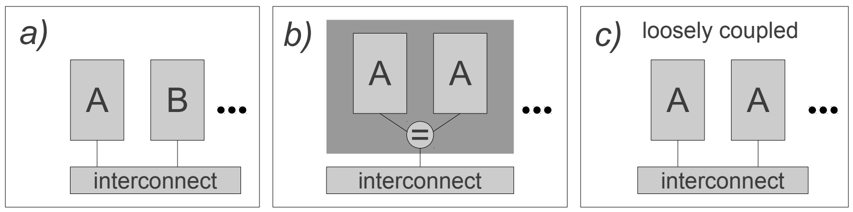


Figure 1: a) non-fault tolerant multi-processor. b) static lockstep processor. c) loosely coupled dynamically coupled processors

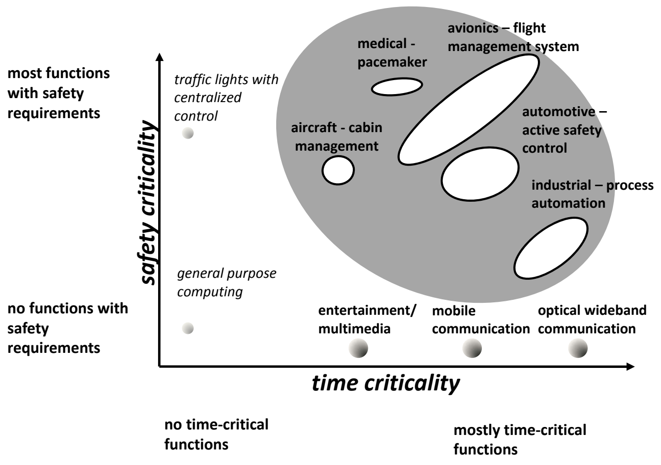
lead to *single-event upsets (SEU)* [Muk08]. A critical charge induced by a charged particle changes the logic-level, which causes a register to flip. An SEU is a transient effect, meaning that the hardware remains fully functional. Flipped bits can occur in all units of modern processors such as memory, register-file or pipeline-control registers. Interestingly, the critical charge required to trigger an SEU, is an inverse proportional function of the supply voltage and the gate-capacitance.

Our research targets embedded systems running mixed-critical applications with varying code complexity, and error handling requirements. We aim to come up with a framework of hardware and software reliability mechanisms from which applications may pick those that are appropriate under a given fault model, timing requirements, and error detection and recovery overhead. For the moment, we focus on transient SEUs in functional units of the CPU. We ignore SEUs at the memory level, as we expect solutions, such as *error-correcting codes (ECC)* [Muk08] implemented in hardware to be available. However, our solution covers faults on the path to and from memory, such as writing to a wrong address or data being corrupted when fetching from memory. Our final vision is to come up with a design that is able to incorporate different hardware fault models and a multitude of resilience mechanisms.

In the remainder of this paper, we take a deeper look at the building blocks to construct ASTEROID, a resilient real-time operating system. We investigate hardware fault models and argue why there is no single solution to resilience that suits the requirements posed by different types of workloads in Section 2. In Section 3 we describe an operating system service that selectively provides transparent redundant multithreading to critical applications. Section 4 introduces hardware-assisted instruction and data fingerprinting, which can be used to speed up software mechanisms and also allows hardening the lower software layers such as the OS kernel itself. We then go on to describe possible ways of implementing recovery in the ASTEOROID system in Section 5 and describe ways for analyzing the real-time behavior of the whole system in Section 6.

2 System, Applications and Fault Models

A wide range of today's embedded application domains, such as smart phones, in-car electronics, and industrial automation, look into consolidating their applications into Multi-Processor Systems on Chip (MPSoC). By using a powerful MPSoC platform, it becomes



most functions with safety requirements
no functions with safety requirements
safety criticality
traffic lights with centralized control
general purpose computing
medical - pacemaker
avionics – flight management system
aircraft - cabin management
automotive – active safety control
industrial – process automation
entertainment/ multimedia
mobile communication
optical wideband communication
time criticality
no time-critical functions
mostly time-critical functions

Figure 2: The two dimensions of criticality: safety and time

possible to integrate computationally intensive algorithms (e.g. camera supported pedestrian detection) with control oriented applications (e.g. active steering).

The architecture of a MPSoC system is in general very similar to the architecture depicted in Figure 1 a). A set of processors is connected to an interconnect facility, which can be a bus, crossbar switch or a sophisticated network-on-chip. Peripheral devices, such as I/O devices and memory controller, are not depicted.

Consolidation into MPSoC systems will integrate different applications with varying requirements into a single system. This system is supposed to allow reliable execution of these applications. A traditional approach to achieve reliability is to statically replicate hardware and software components, for instance by performing static lock stepping as depicted in Figure 1 b). However, this static approach requires resource over-provisioning, because it replicates the whole software stack instead of taking into account the specific requirements of each application.

Some applications may have hard real-time constraints or are safety-critical whereas others are non-critical at all (e.g. best-effort entertainment). A system design incorporating such considerations is said to support *mixed-criticality* applications. *Criticality* can be broken down into two orthogonal dimensions as depicted in Figure 2.

For time-critical applications without safety requirements, the sole focus lies on timeliness of computation or communication. Typical examples for this domain are embedded mobile communication applications e.g. UMTS or LTE, as well as video decoding. Here, guaranteed data integrity is not safety-relevant because transient or permanent error conditions do not have catastrophic consequences.

For purely safety-defined systems it is crucial that integrity of computation is preserved, even in presence of errors, e.g. traffic lights which are controlled by a centralized control facility. In case of failures, pedestrians and car passengers can be injured or even killed, thus safety demands are high. However, it is not critical if light phases are of accurate timing, as long as the signals are consistent. The taxonomy of safety is defined by safety standards and is the focus of dependability research [ALRL04].

Additionally, some application scenarios require both, timing and safety constraints, to be met. Examples in this area are in-car safety systems, such as airbag controllers. From a reliability perspective, such applications demand an extremely short fault detection latency. For recovery, a simple scheme, such as restarting the application from an initial checkpoint, may suffice.

It appears there is no one-size-fits-all solution that can be applied to all of the above scenarios. Therefore, in the ASTEROID system architecture we try to accommodate the needs of all applications by providing a set of detection and recovery techniques from which developers can chose the ones that best fit their requirements in terms of latency, energy consumption, or normal-mode execution overhead. Our approach is to use loosely coupled cores as depicted in Figure 1 c). These cores can either be used to replicate critical applications, or they can be used as single cores to execute applications that are not replicated because they either are not critical enough or they leverage other means of fault tolerant execution, such as arithmetic encoding [FSS09].

The operating system plays a crucial role in our resilient design. On the one hand it needs to provide error detection and recovery mechanisms in those cases where hardware mechanisms don't suffice. On the other hand, studies show that many kinds of hardware errors finally also lead to errors that corrupt operating system state [LRS+08, SWK+05]. Our work uses the L4/Fiasco.OC microkernel [LW09] as the OS basis. The inherent design of the microkernel provides us with fine-grained isolation between OS components. As we can expect more than 65% of all hardware errors to corrupt OS state [LRS+08] before they are detected, this isolation allows us to limit the propagation of corrupted data and therefore promises to reduce error detection and recovery overhead.

3 Redundant Execution as an Operating System Service

In general, fault-tolerant systems deal with potential hardware faults by adding redundancy. Such redundancy can for instance be added by using duplicate hardware execution units [BBV+05, IBM08], employing redundant virtual machines [BS95], or compiling the software stack with compilers that add redundant computations [RCV+05] or apply arithmetic encodings to data and instructions [FSS09].

Compiler-based replication also allows to replicate at an application granularity. However, it requires the source code of all applications to be available for recompilation, which is not true for many proprietary applications. For instance, best-effort applications for mobile devices often come from popular app stores, where the user has no influence at all on how the applications are compiled.

As a result, for ASTEROID we decided to provide replicated execution as an operating system service on top of the L4/Fiasco.OC microkernel. The resulting system architecture is shown in Figure 3. The base system is split into an operating system kernel running in privileged mode and additional runtime services running as user-level components. To

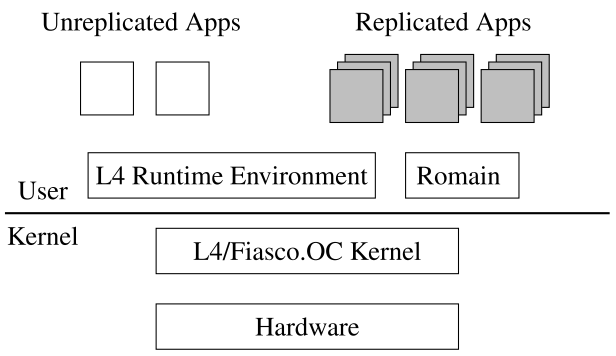


Figure 3: ASTEROID system architecture

this system we add a new component, *Romain*, which provides software-implemented redundant multithreading [RM00].

The Romain framework provides replication for critical user-level applications without requiring changes to or support from the replicated application. Furthermore, as we are basing our work on a microkernel, complex OS code, such as device drivers and protocol implementations, runs in user-level components which can be covered by this replication scheme as well.

The current implementation of Romain allows replicating single-threaded applications on top of L4/Fiasco.OC. Romain splits the application into N replicas and a master that manages and controls these replicas as depicted in Figure 4. The master serves as a shim between the replicas and the rest of the system. It makes sure that all replicas see the same input (IPC messages, results from certain instructions, such as reading the hardware time stamp counter) at the same point in time, because only then can we enforce deterministic behavior in all replicas.

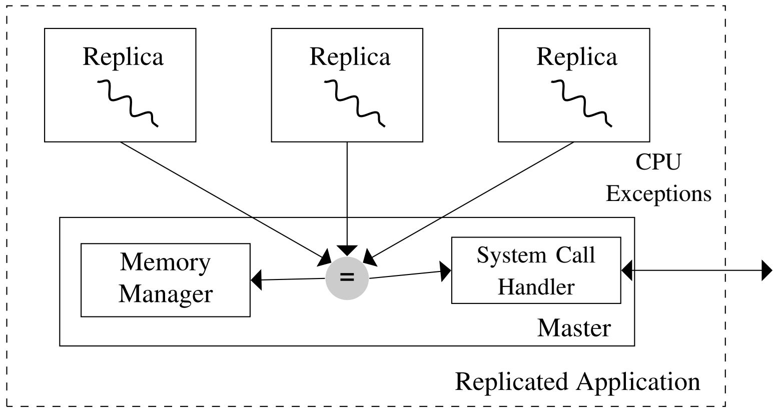


Figure 4: Romain Architecture

To prevent undetected faults from propagating into other replicas before the master detects them, each replica is executed within a dedicated address space. The master process leverages L4/Fiasco.OC vCPUs [LWP10], a mechanism originally designed to implement virtual machines, to launch the replicas. The benefit of using a vCPU is that whenever a replica raises a CPU exception, such as a system call or an MMU fault, this exception automatically gets reflected to the master process.

Once a CPU exception is raised, the master waits for all replicas to raise their next exception and then compares their states. This comparison includes the replicas' CPU registers, as well as kernel-provided control information, such as the type of the raised exception. Furthermore, the L4/Fiasco.OC User-Level Thread Control Block (UTCB), a memory region used to pass system call parameters to the kernel, is inspected for each replica. The state comparison does not include a comparison of the complete replica address spaces as this would lead to infeasible runtime overheads.

The decision to not incorporate all memory into the state comparison manifests a trade-off between lower runtime overhead and lower error detection latency. In Section 4 we present instruction fingerprinting, a small hardware extension that will help us to achieve a low runtime overhead while incorporating the complete state of a replica into the required state comparison.

In addition to performing state comparisons, the master process manages the replicas' memory layout. As explained in Section 1, we assume memory to be protected by ECC. This means that once data is stored in memory, we can assume it to remain correct. Hence, any read-only memory can be assumed to never be corrupted. Therefore, such memory regions only exist as a single copy and are shared among all replicas. In contrast, each replica works on a private copy of every writable memory region.

The memory management strategy discussed above works for memory that is local to the replicated application. However, applications in the ASTEROID system may also use shared memory for transmitting data back and forth. In a setup involving replicated applications, the master then needs to make sure that whenever replicas access shared memory, they read/write the same data regardless of timing or order of access. Hence, in contrast to application-internal memory, the master needs to intercept and emulate all accesses.

Emulating memory accesses would involve an instruction emulator as part of the master process, which has drawbacks in terms of runtime overhead, portability, and maintainability: the emulator adds a large amount of code to the master process, emulation is slower than directly executing the instructions on the physical CPU, and for every hardware platform the system is supposed to run on, the emulator needs to be implemented from scratch.

To avoid the complexity of a full instruction emulator, we implement a copy&execute technique in Romain. As a prerequisite for this, the master makes sure that shared memory regions are mapped to the same virtual address. However, only the master gets full read/write access to the region, whereas the replicas raise page faults upon each access. When encountering a page fault, the master inspects the replica state and copies the faulting instruction into a buffer local to the master. This buffer contains code to a) push the replica's CPU state into the master's physical registers, b) execute the faulting instruction, and c) restore the replica's and master's register states. The master then executes the code within this buffer.

The copy&execute strategy works for most shared memory accesses, but has two main drawbacks: First, a few instructions are unsupported (e.g., indirect memory jumps, and instructions with multiple memory operands). Second, the execution of the instruction within the master is not replicated. Therefore, a transient fault that occurs while executing this instruction will not be detected. In future work we will explore ways to address these issues.

It remains to be noted, that while Romain protects a wide range of user-level components against hardware errors, it does not protect the microkernel itself nor the Romain master process. Furthermore, it relies on certain hardware mechanisms (exception delivery, address space protection, MMU fault handling) to function correctly. We refer to this set of components as the *Reliable Computing Base (RCB)* [ED12], which needs to be protected in a different way. We are aware of the problem and believe that the hardware mechanisms described in Section 4 can serve as initial building blocks for solving it. However, this is still an open issue and left for future work.

To evaluate the overhead imposed by replicating applications using Romain, we ran experiments using the MiBench benchmark suite as a single instance, as well as in double- and triple-modular redundancy mode. We compared their runtimes on a test computer with 12 physical Intel Core2 CPUs running at 2.6 Ghz. Hyperthreading was turned off and every replica as well as the master were pinned to a dedicated physical CPU.

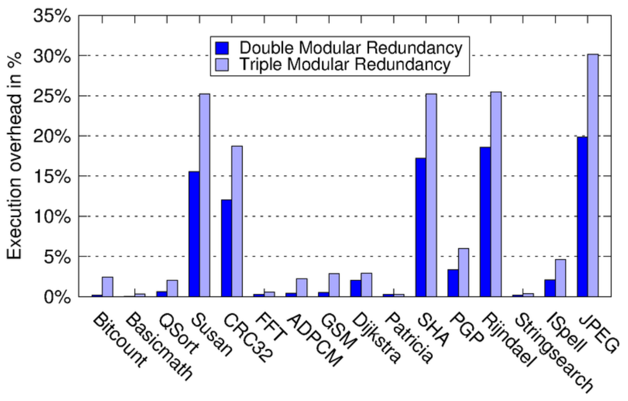


Figure 5: Romain overhead for MiBench benchmarks in double and triple modular redundancy mode

The results in Figure 5 show that the normalized runtime overheads range between 0.5% and 30%, but are below 5% in most of the benchmarks. Further investigation showed that the overheads correlate with the amount of memory faults raised by the benchmark – more page faults lead to more interaction with the master process and therefore imply a higher runtime cost. In future work we will therefore also investigate whether the memory management cost can be optimized.

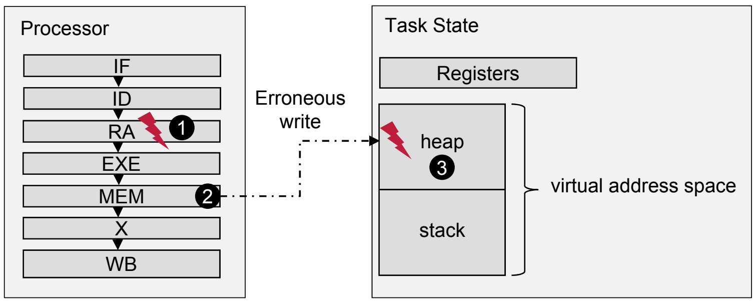


Figure 6: Illustrative example of an error in the processor pipeline (1) which causes an erroneous write (2) and leads to an error in the heap state (3).

4 Error Detection Using Hardware-Assisted Fingerprinting

A central aspect of fault-tolerance is the employed error detection scheme, as already discussed in the previous sections. Figure 6 shows how pipeline errors propagate from the processor into the task's state. Here the task state consists of the entire virtual memory space as well as the architecturally visible registers.

For instance an illegal register access causes an erroneous operand fetch (1). When the content of this register is used later for memory accesses (e.g. writes), the state of the task is modified illegally. The objective of error-detection is to identify and signal such alterations. There are several metrics by which we can measure the quality of the error-detection mechanism:

- *Error coverage*, which is the fraction of errors which are detectable by the mechanism, should be as high as possible
- *Error latency* which is the time from error occurrence to error detection, should be as low as possible.
- *Additional overhead* (performance penalty, chip area, code size) should be as low as possible.

The Romain architecture as presented in Section 3 compares state on externalization only (e.g. on system-calls and exceptions). Thus, the error coverage of Romain is sufficiently high, because all data is eventually subject to a comparison before it becomes visible. As already discussed, without further consideration of shared-memory communication the execution time overhead of the presented approach is reasonably low. But it also comes with some inherent drawbacks with respect to our requirements: The major issue is that the error latency is not bounded. An error in the task's state as depicted in Figure 6 can stay dormant for long time until the erroneous state is externalized. An arbitrary long detection latency can potentially render an error recovery mechanism useless if real-time requirements are involved.

To circumvent this problem we use hardware assisted fingerprinting, which was introduced in [SGK^{+}04]: A dedicated fingerprint unit which resides in the pipelines of all cores in

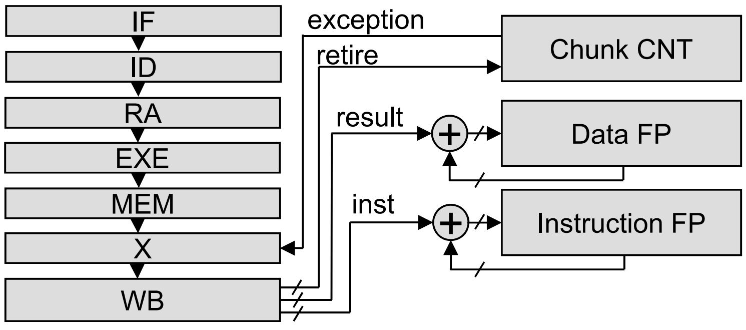


Figure 7: Leon 3 pipeline with fingerprinting extensions.

the processor hashes all retired instructions. This generates a fingerprint which represents an unique hash for a specific instruction/data sequence. Since the same code is executed on redundant cores we can use the fingerprint as a basis for DMR voting. In the original work from [SGK$^+$04], voting between redundant cores is performed when cache lines become visible on the system bus. However, this approach has some inherent drawbacks, especially in the field of real-time systems and with respect to mixed-critical applications. Since the mechanism relies on the cache coherency protocol as a synchronization primitive for comparison, the mechanism implicates a high degree of timing uncertainty (e.g. when comparisons are performed and how often). Also, no differentiation between task contexts is made, thus all instructions end up in one single fingerprint and redundancy cannot easily be performed task-wise.

Thus, we propose to use fingerprinting differently and implemented context-aware fingerprinting, where a fingerprint is generated per context (if required). We extended the LEON 3 processor [SWSG06] with a fingerprint unit as shown in Figure 7. The unit consists of three building blocks: a chunk counter which counts retired instructions, the data fingerprint which taps the data path of the pipeline and the instruction fingerprint which is fed with the retired instruction word. All of these registers are implemented as ancillary state registers (ASRs) which can be read by software.

The unit works the following way: Both fingerprint registers continuously hash data and instructions. In case of interrupts or traps, the processor will store a copy of the recent fingerprint and the operating system may store the fingerprint in the task control block. In the same way an old fingerprint can be restored on a return-from-interrupt instruction. Thereby, per-task fingerprints can be implemented by the operating system and we are able to handle asynchronous events.

Data and instruction fingerprint reflect a hash over the task state and can be used in the Romain master for voting. However, this approach still exhibits the drawback of an unbounded detection latency, because a task first needs to raise a CPU exception to trigger comparison.

To artificially increase the voting-frequency in a predictable way, we implemented chunk checking. Chunk Checking is a feature which is controlled by the operating system to control the error detection latency for long-running workloads. Per se, the operating system has no method to interrupt two copies at a predictable instant in time (on exactly the same instruction) in order to compare intermediate results. Here, we use the chunk counter which is decremented with each executed instruction and causes a trap if it reaches zero. This enables the operating system to compare intermediate results without using the highly inefficient single-stepping mode.

A third mode of operation is the signature checking mode. In this mode we leverage from the fact that we have an individual instruction fingerprint. By construction it is possible to pre-compute instruction fingerprints for each basic block. This can be done by the compiler or by dynamic recompilation during runtime as part of an operating system service. This enables to implement near zero-overhead signature checking for basic blocks: A dedicated match-fingerprint instruction tests the target and the actual fingerprint which may result in a fingerprint-miss trap.

5 Recovery

The mechanisms described in this paper allow us to detect transient hardware errors. However, a fully functional system also needs to recover from these errors. In this Section we present ideas on ways to provide recovery in the ASTEROID system. As explained in Section 3, our system supports arbitrary n-way modular redundancy. This implies that we can use voting among the replicas to find the set of correct replicas and then overwrite the faulting replicas states with correct ones.

Unfortunately, from a practical perspective every additional layer of redundancy will increase the runtime overhead implied by our system and therefore users may opt to use double-modular redundancy (DMR). This approach allows only error detection, but does not include recovery. Therefore, DMR needs to be combined with a checkpoint/restart mechanism. We are currently investigating state if state-of-the-art checkpointing such as [AAC09] is suitable to fit into ASTEROID.

Restarting from a checkpoint has implications on the rest of the system, because other applications may depend on the service provided by the restarted application and the restart may invalidate recent computations or application service state. This problem has been investigated before [Her10, DCCC08]. Our solution to reintegration is based on L4ReAnimator [VDL10], an application framework that keeps track of references to external services.

L4ReAnimator is responsible for a) storing how these references were obtained, and b) monitoring the references in order to detect service interruption. Once an external service is restarted, the framework will lazily detect this restart upon the next invocation of the service. Then, it will re-establish a connection using the stored connection information and potentially recover a consistent session state. Thereafter, the application may continue using the service.

In addition to combining ASTEROID's error detection with L4ReAnimator, we plan to investigate whether we can use heuristics to figure out the faulty replica in a DMR setup. The first heuristic could be based on observing CPU exceptions raised by the replicas: assuming that transient faults often lead to crashes, we will see a faulty replica raise a CPU

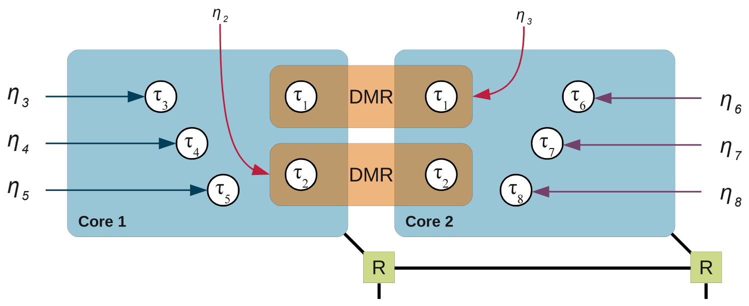


Figure 8: Example for task level redundancy on a MPSoC

exception (e.g., an invalid memory access) while the non-faulty replica will simply continue execution and raise no fault. In such case, we might deduce that the faulting replica is the broken one. However, we cannot be 100% confident, because the invalid memory access may well be valid application behavior and the faulty replica may simply be stuck in an infinite loop raising no exceptions. Initial fault injection experiments indicate that such infinite loops are rare exceptions, though. Furthermore, the chunk checking mechanism described in Section 4 will force any replica to raise an exception within a bounded amount of time.

Another useful heuristic may involve programmer interaction. Applications may come with built-in consistency checks in the form of assertions added to the application's code. When encountering a fault, the master process may then check if the fault was raised by a known assertion from the application code and thereby deduce the faulty replica.

Another option would be to add consistency checker functions. These can be registered with the master process upon startup. Upon a fault the master can then run the checker function on the replicas' states to detect the faulty replica. As the checker function includes developer knowledge, it may be much faster in figuring out which replica's state is inconsistent.

6 Can we Provide Real-Time Guarantees?

The correctness of behavior depends not only on the logical result, but especially in time-critical systems, also on the instant in time at which the result is produced [Kop97]. So even if the system is capable of detecting and recovering from errors on the fly, it is still possible to miss deadlines caused by transient overload effects. Thus for real-time systems we must differentiate between logical correctness and timing correctness. The system is working correctly only if the platform satisfies two criteria:

1. the result must be logically correct according to the specification
2. the data must be delivered in time (i.e. deadlines must be met)

Vise versa, a *failure* is defined as either logical incorrectness or a timing violation.

The problem of ensuring logical correctness can be solved by replicating only critical tasks as shown in Figure 8. This particular example shows two cores which are part of a larger MPSoC. Some tasks of the task graph execute safety-critical code (τ_1, τ_2), the other represent other applications (e.g. uncritical, hard real-time or best-effort) computations. As annotated, safety critical tasks are executed redundantly in a DMR fashion on both cores. Given that the operating system performs comparison operations between all external I/O issued by τ_1 and τ_2, the platform is capable of detecting logical errors for critical tasks without massive duplication.

With respect to timing violations, the first question that needs to be raised is "why do we need novel real-time analysis methods?" Former research in the field of performance analysis led to a sophisticated set of analysis approaches. Most of them are based on the well studied busy-window approach initially introduced in [Leh90], which allows us to analyze component behavior. We can use it to derive the response-time, which is the largest time from task activation until the data has been processed. If the response-time is smaller than a specified deadline, then this task is schedulable.

The busy window idea can also be extended to a system-wide analysis which considers communicating tasks. Both, compositional performance analysis (CPA) [HHJ+05] and modular performance analysis (MPA) [Wan06], provide such functionality.

Unfortunately, CPA and MPA alone are not capable of analyzing error events and recovery operations, because their effects are not bounded in time. Although unlikely, it is generally possible that a lot of errors happen at once which leads to a violation of real-time properties. Thus, since error occurrences are probabilistic we must transition to a probabilistic real-time analysis.

Safety regulations dictate a minimal level of dependability for safety critical functions. Therefore, most standards define a probabilistic metric such as Mean Time To Failure (MTTF) in order to quantify reliability. As software function is generally implemented as a task graph which is mapped to computational and communication resources, those resources inherit the software's dependability requirements. Thus, the following analysis must be able to show that the actual MTTF of a task is larger than the required MTTF.

We formulated a timing analysis which accounts for errors and recovery events in [ASE11]. Due to limited space, in this paper we only sketch the idea briefly. For the mathematical description we refer to [ASE11].

Our analysis focuses on transient errors and soft errors in particular. Soft errors caused by radiation or variability will manifest as logical bit-flips in architectural registers. We assume that the fault causing bits to flip will vanish immediately and has no temporal extent (there are no intermittent errors). Since the exact arrival of error-events is highly unpredictable, we use stochastic models to evaluate the behavior on a probabilistic base. The occurrence of errors on a core is modeled using Poisson processes with a given error-rate λ_i per core which accounts for errors affecting core components (e.g. ALU and FPU). Specifying per-core error rates models heterogeneous multi-core CPUs in which some cores may inherently be more reliable than others. This can be the case if dedicated hardware error detection and recovery mechanisms are used, such as proposed by [ABMF04] or by using more reliable but less performant silicon process parameters.

Given our model, the following equation gives the probability for unaffected execution of processor P_i during the time interval Δt:

$$P(\textit{no error in time } \Delta t) = e^{-\lambda_i \Delta t} \tag{1}$$

To derive the reliability for each task, we use a two step algorithm. First, we identify possible error manifestations (scenario) which lead to feasible schedules. Therefore, we model the specific timing-effect of the error-scenario and analyze the model using response-time analysis methods, known from the CPA.

In the second step, after discovering the working scenarios, we can use Equation 1 to calculate the probability that one working scenario will actually happen. Based on these probabilities it is possible to calculate the MTTF in order to decide whether the reliability is sufficiently high. This in turn lets us decide whether tasks under errors met their deadlines sufficiently often to fulfill all requirements.

7 Conclusion

In this paper we presented building blocks for ASTEROID, an analyzable, resilient, embedded operating system design. ASTEROID accommodates applications with varying safety, timing, and resource requirements. Unreliable applications are hardened by using transparent software-implemented redundant multithreading. A fingerprinting mechanism in hardware decreases error detection latencies and increases error coverage. Additionally, we sketched ideas on how to perform error recovery and gave a brief overview of how the ASTEROID system can be analyzed with respect to real-time requirements in the presence of transient hardware errors.

Acknowledgments

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

References

[AAC09] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[ABMF04] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with Razor. *IEEE Computer*, 37(3):57–65, 2004.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, 2004.

[ASE11] Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability Analysis for MPSoCs with Mixed-Critical, Hard Real-Time Constraints. In *Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Taiwan, October 2011.

[BBV$^+$05] D. Bernick, B. Bruckert, P.D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop: Advanced Architecture. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 12–21, june-1 july 2005.

[Bor05] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10 – 16, nov.-dec. 2005.

[BS95] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[DCCC08] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 59–72, San Diego, CA, December 2008.

[ED12] Michael Engel and Björn Döbel. The Reliable Computing Base – A Paradigm for Software-Based Reliability. In *Proc. of the 1st Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'12, 2012.

[FSS09] Christof Fetzer, Ute Schiffel, and Martin Süsskraut. AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 283–296, Berlin, Heidelberg, 2009. Springer-Verlag.

[Her10] Jorrit N. Herder. *Building a dependable operating system: Fault Tolerance in MINIX3*. Dissertation, Vrije Universiteit Amsterdam, 2010.

[HHJ$^+$05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The SymTA/S Approach. *IEE Proc. Computers and Digital Techniques*, 152(2):148–166, March 2005.

[IBM08] IBM. PowerPC 750GX Lockstep Facility. IBM Application Note, 2008.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Leh90] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proc. 11th Real-Time Systems Symposium*, pages 201–209, Dec 1990.

[LRS$^+$08] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

[LW09] Adam Lackorzynski and Alexander Warg. 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*, pages 25–30, Nuremberg, Germany, 2009. ACM.

[LWP10] Adam Lackorzynski, Alexander Warg, and Michael Peter. Generic Virtualization with Virtual Processors. In *Proceedings of Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, October 2010.

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Nas10] Sani R. Nassif. The light at the end of the CMOS tunnel. In *Int. Conf. on Application-specific Systems Architectures and Processors*, pages 4 –9, july 2010.

[RCV$^+$05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254. IEEE Computer Society, 2005.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News*, 28:25–36, May 2000.

[Sch07] Dieter K. Schroder. Negative bias temperature instability: What do we understand? *Microelectronics Reliability*, 47(6):841–852, 2007.

[SGK$^+$04] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatryk. Fingerprinting: bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6):22–29, 2004.

[SWK$^+$05] Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*, 25:30–39, November 2005.

[SWSG06] Zoran Stamenkovic, C. Wolf, Günter Schoof, and Jiri Gaisler. LEON-2: General Purpose Processor for a Wireless Engine. In Matteo Sonza Reorda, Ondrej Novák, Bernd Straube, Hana Kubatova, Zdenek Kotásek, Pavel Kubalík, Raimund Ubar, and Jiri Bucek, editors, *DDECS*, pages 50–53. IEEE Computer Society, 2006.

[VDL10] Dirk Vogt, Björn Döbel, and Adam Lackorzynski. Stay strong, stay safe: Enhancing Reliability of a Secure Operating System. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS 2010)*, Paris, France, April 2010. ACM.

[Wan06] E. Wandeler. *Modular performance analysis and interface-based design for embedded real-time systems*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 2006.

[ZMM04] Dakai Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, ICCAD '04, pages 35–40, Washington, DC, USA, 2004. IEEE Computer Society.