Where Have all the Cycles Gone? Investigating Runtime Overheads of OS-Assisted Replication

Björn Döbel, Hermann Härtig

Operating Systems Group
TU Dresden
Nöthnitzer Str. 46
01187 Dresden
{doebel,haertig}@tudos.org

Abstract: In order to allow user-level applications tolerate transient hardware faults, we developed *Romain*, an operating system service that transparently replicates unmodified binary applications. While replication increases overall system reliability, it also requires additional resources and runtime. In this paper we evaluate *Romain*'s runtime overhead using the SPEC INT 2006 benchmark suite. With most of the benchmarks being compute-bound they lend themselves to low overhead replication and the geometric mean of their runtime overhead for triple-modular redundant execution is only 1.8%.

More surprisingly, during our measurements we also encountered issues not directly related to replication. We show that improper placement of replicas to CPU cores as well as unoptimized use of memory management mechanisms can make a significant contribution to runtime overhead and discuss how *Romain* avoids these pitfalls. We finally use our measurement results to model how protecting the Reliable Computing Base using compiler-based fault tolerance mechanisms impacts replication overhead.

1 Introduction

Decreasing feature sizes following the predictions of Moore's law are a driving factor for innovation at the hardware level. They empower hardware vendors to integrate more and more cores and functional units into the same physical dies, thereby enabling higher levels of performance and features. However, these smaller hardware features are more vulnerable to transient hardware errors that arise due to hardware aging, thermal effects, as well as cosmic radiation [Bor05]. These effects were studied in the field: Schroeder et al. found that up to 8% of DRAM DIMMs in a large-scale server installation suffered from soft errors within a year [SPW09]. Fiala et al. estimate that state-of-the-art reliability measures in High-Performance Computing may require 65% of the available compute time for checkpoint/restart instead of performing useful computations [FME+12].

Of course, hardware vendors try to implement hardware that meets certain reliability targets even at lower levels of integration. This is usually achieved by adding hardware-level reliability extensions, as for instance proposed by DIVA's checker cores [Aus99]. These

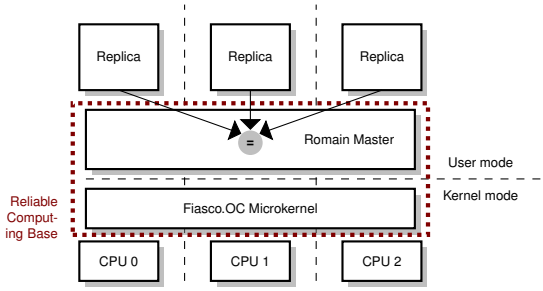

Figure 1: Replicated execution using *Romain*

additional units mask faults at the hardware level so that software can run oblivious of potential hardware malfunctions. Unfortunately, these solutions are often too expensive to be applied within commercial-off-the-shelf (COTS) systems. From a research perspective it is therefore interesting to determine if software-level solutions can help hardware vendors meet their goals.

COTS machines rely on software to take care of error detection and correction itself. Compilers generating fault-tolerant code, such as SWIFT [RCV+05] and AN-Encoded Processing [FSS09] use instruction duplication or arithmetic encoding of data and execution paths to detect errors without requiring special hardware support. However, being part of a compiler, these tools require access to the source code of all applications that should be protected. They cannot deal with binary-only third-party software as it is for instance provided through internet downloads and mobile phone appstores.

To support resilient execution without having access to the application's source code, we implemented *Romain*, an operating system service on top of the L4/Fiasco.OC microkernel[1] that replicates binary applications [DHE12] using a software implementation of redundant multithreading [RM00]. As shown in Figure 1, *Romain* consists of a master process that manages multiple replicas of a binary application. The master loads the binaries into separate address spaces to facilitate fault isolation.

Replicas execute independently on the physical CPU as long as they only modify their internal state (e.g., perform computations and read/write their private address space). For now we assume replicas to be single-threaded, which means that as long as they execute the same code on the same set of inputs, they will deterministically expose identical behavior [Sch90].

Romain intercepts replica execution once the replicas try to make their internal state externally visible by performing a system call or raising hardware traps, such as page faults. These *externalization events* get redirected to the master. The master then waits for all replicas to reach their next externalization event and compares their states (architectural register file and Fiasco.OC's user-visible per-thread state). A watchdog mechanism is used to make sure the master does not wait infinitely, e.g, because of a faulty replica stuck in an infinite loop [Kri13].

[1] `http://www.tudos.org/fiasco`

If the replicas' states match, the master performs the respective system call or resource management operation on behalf of the replicas and thereafter resumes independent execution of the replicas. Upon a state mismatch, the master triggers recovery. *Romain* supports different ways of recovering from faults. When running in triple-modular redundant mode (TMR), the master performs majority voting and overwrites the faulty replica. If the user wants to save on required resources by only using two replicas (DMR mode), the master can also trigger reset to a previous application checkpoint.

For replicas to behave deterministically, *Romain* needs to enforce that they all observe the same inputs. For this purpose the master process intercepts all potentially non-deterministic inputs that reach the replicas. In Fiasco.OC most input arrives through synchronous IPC system calls, which are intercepted and checked by the master anyway. Further non-determinism is induced for instance by reading the CPU's time stamp counter. Such operations can be intercepted by the *Romain* master using software interrupts (`INT 3` on x86) or by exploiting virtual machine extensions in modern hardware [Int13]. Scheduling in multithreaded applications is a last source of non-determinism. While we are currently working on a solution to this problem, we only consider single-threaded applications for our analysis in this paper.

Replicated execution transparently increases applications' fault tolerance. However, this reliability comes at the cost of increased resource demand and execution time. This paper focusses on analyzing the runtime overhead caused by replication. We evaluate *Romain* using the SPEC INT 2006 benchmark suite. We introduce our experiment setup in Section 2 and describe specific effects caused by placing replicas on different sets of cores in Section 3. Section 4 discusses the sources of overheads for replicating SPEC INT 2006.

While *Romain* protects user-level applications from hardware faults, the master process as well as the underlying microkernel are still left unprotected. We call these components the *Reliable Computing Base (RCB)* [ED12]. The RCB needs to be protected from hardware errors separately, for instance by compiling RCB components using a fault-tolerant compiler. In Section 5 we use our benchmark results to estimate the impact that hardening the RCB using compiler-based methods would have on *Romain*.

2 Experiment Setup

For our experiments we use the SPEC INT 2006 benchmark suite [Hen06]. We ported 11 out of the 12 benchmarks to Fiasco.OC's L4Re runtime environment[2]. We left out the `483.xalancbmk` because it uses deprecated C++ STL features that are not supported by L4Re's C++ standard library. SPEC INT is a CPU benchmark suite, which means the benchmarks represent replication-friendly applications where replicas perform lots of independent computation and only rarely suffer from overhead for state comparison. Nevertheless, the benchmark suite represents real-world use cases, such as compilers, dynamic languages, and video decoding.

Our evaluation computer comprises 3 GB of RAM and two processor sockets, each containing 6 Intel Xeon X5650 CPUs clocked at 2.667 GHz. Fiasco.OC, the L4Re runtime, and the SPEC INT benchmarks were compiled for 32-bit execution, because *Romain* cur-

[2] `http://www.tudos.org/l4re`

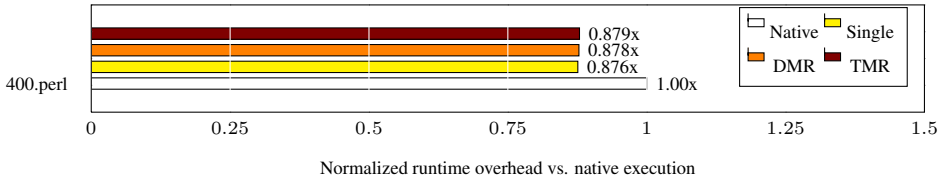

Figure 2: Overhead for `400.perl`, first experiment

rently only works for 32-bit binaries. We turned off hyperthreading to minimize side effects, which left us with 12 physical CPU cores to run the benchmarks on. We also turned off turbo-boost and hardware-level frequency scaling to get stable execution times for the benchmarks.

To get a performance baseline, we ran the benchmarks on a Fiasco/L4Re base system consisting of the kernel and the user-level resource managers. To obtain reproducible results, we did not run any other workloads on the system while performing our measurements. We refer to these baseline measurements as *native execution* in the remainder of this paper. We report normalized overheads for *Romain* running a single replica. This allows us to evaluate the overhead caused by the master's system call interception and replica resource management. Furthermore we report normalized overheads for fault-tolerant execution with *Romain* running two (DMR) and three (TMR) replicas. For multi-replica runs we pin each replica thread to a dedicated physical CPU core to get the least possible interference between the replicas.

3 Core Placement – Doing it Right?

With this setup in place we started benchmarking with the `400.perl` benchmark. The results of our measurements are shown in Figure 2. We expected replication to slow execution down. Now, to our very surprise the benchmarks were running about 12% faster! We repeated the benchmark run several times and the results remained the same. Furthermore, these 12% amounted to a wall clock time difference of about 30 seconds for this benchmark, which rules out measurement inaccuracies.

We then suspected cache effects to be responsible for our observations. To validate this, we used the processors' built-in performance counters and measured the `ITLB_MISSES`, `DTLB_MISSES`, `L2_MISSES`, and `L3_MISSES` performance counters for every CPU core participating in these runs. If the caches were responsible for this strange behavior, the number of cache misses for a *Romain* run should be lower than the one for native execution. However, it turned out that these numbers were nearly exactly the same and hence, caches were not the problem here.

Exploring the available parameters, we noticed that the native run got faster if we executed it on any physical CPU except CPU 0. This hinted at a configuration problem. Fiasco.OC is a microkernel and therefore requires a couple of other components to run in user space, even if we only execute a single benchmark as our workload. One of these components is the logging component responsible for printing output written by functions such as `printf`. The `400.perl` benchmark prints several thousand lines of data, but

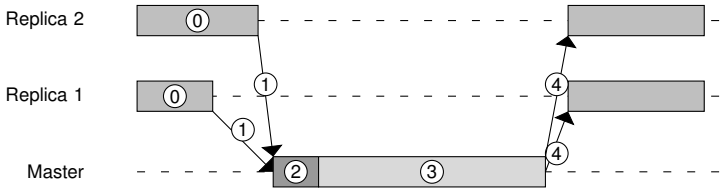

Figure 3: Replication: Fault Handling. (0) - Replicas execute independently. (1) - Externalization event is propagated to the master. (2) - Master compares replica states. (3) - Master handles system call. (4) - Master returns control to the replicas.

our intuition told us this should not be a problem: as both the native and replicated runs perform these operations, the time for the respective system calls should be identical.

A closer look at how output works on Fiasco.OC taught us differently. Calling the C library's `printf` function results in an IPC message being sent to the log thread. This higher-priority thread is part of the runtime's log server and by default executes on CPU 0. The log thread receives this message and then prints it. The default method of printing is to send this output to the serial console, which in turn is rather slow.

If the benchmark runs on CPU 0, as was the case for our native measurement, the log message causes the log thread to occupy the CPU for the whole duration of the output operation due to its higher priority. This prevents the benchmark application from making further progress. However, if we move the benchmark to a different CPU, it is only blocked until the log IPC message was delivered. Thereafter, both threads can execute concurrently and therefore the benchmark runs faster. To our slight humiliation, the reason for *Romain* performing faster in our initial measurement was merely that the *Romain* master starts placing replicas on CPU 1 and therefore coincidentally avoided being slowed down by the log thread.

The lesson to take away from this experience is: Small differences in system configuration (e.g., placement of threads not directly related to our benchmark application) can have a significant impact on performance. We will see in Section 4 that the 12% difference we observed here is much higher than the real replication-induced performance. The measurements in the remainder of this paper avoid logging slowdown by sending their output to an in-memory buffer similar to redirecting output to a file on Linux. The corrected results for `400.perl` are shown in Section 4.

Motivated by the previous measurement we also re-evaluated how assigning replicas to CPUs influences replication overhead. Figure 3 shows how a system call is handled by the *Romain* master for the case of two replicas. Both replicas execute independently (0) until they raise an externalization event on their local CPU (1) and block. If one replica lags behind the other, some wait time may arise. Next, the event is propagated to the master, which then performs checksum computation (2) and handles the system call (3). Finally, the master returns control to the replicas (4).

We aim to produce an ideal environment for our experiment and do not run any other applications on our system during benchmark runs. Hence, the wait time that arises at externalization is negligible. Furthermore, the cost of system call handling (step 3) is in

most cases identical to native execution, because the master simply performs the system call that would have been issued by the application otherwise.

Replication overhead stems from steps 1, 2, and 4. We used a microbenchmark to estimate the cost of these steps. Checksum computation is a simple sum operation and takes about 100 CPU cycles per replica. This is rather fast compared to the notifications sent in steps 1 and 4. By default, Fiasco.OC's virtual CPU mechanism [LWP10] is used for sending an event and the corresponding replica state to the master. On our test machine, a plain vCPU exception costs about 2,200 CPU cycles when replica and master execute on the same CPU. As *Romain* replicas run on dedicated cores, event notifications need to be sent across CPUs, which adds expensive inter-processor interrupts (IPIs) to the number. In our two-socket system we need to additionally distinguish between IPIs sent within a single socket and those sent across socket boundaries. We measured the average cost of an exception sent across CPUs to be 5,900 cycles for intra-socket communication and 14,300 cycles for cross-socket messages.

Based on these microbenchmarks we implemented a core placement algorithm in *Romain* that prefers to assign replicas to CPUs on the same socket, because thereby we save about 8,000 cycles (or 60%) for steps 1 and 4 as opposed to distributing replicas across sockets. The results reported in the next section were obtained using this placement algorithm.

4 Replicating SPEC INT 2006

After overcoming the previously discussed hurdles, we ran all 11 of the SPEC INT 2006 benchmarks. Figure 4 shows the runtime overheads normalized against native execution. The geometric mean overhead for running three replicas is 2.51%. This matches our initial expectations, because the benchmarks are mostly CPU-bound and these computations can happen concurrently on the replica's cores while overhead for externally visible events only occurs rarely. Four benchmarks, `403.gcc`, `429.mcf`, `462.libquantum`, and `471.omnetpp` experience higher overheads. Therefore, we focussed our further investigations on these applications.

The `403.gcc` benchmark carries out a large amount of memory remap operations (using the C library's `mremap` and `realloc` functions). This behavior stresses *Romain*'s memory management capabilities by requiring a large amount of allocations and page fault handling. We actually ran out of memory for four out of the nine compiler runs when running with three replicas. Furthermore, the benchmark also triggers some bugs in *Romain*'s memory manager, which is why we take these numbers with a grain of salt.

4.1 Improving Replica Memory Management

Observing that memory management appears to be a bottleneck, we revisited *Romain*'s memory management. The master process always keeps one copy of every memory region per replica and uses Fiasco.OC's user-level memory management mechanisms to manage the replicas' address spaces based on those copies. On 32-bit Fiasco.OC, every address space can contain 3 GB of user-addressable memory. As a consequence, *Romain* can only run three replicas as long as a single replica does not use more than 1 GB of memory. This

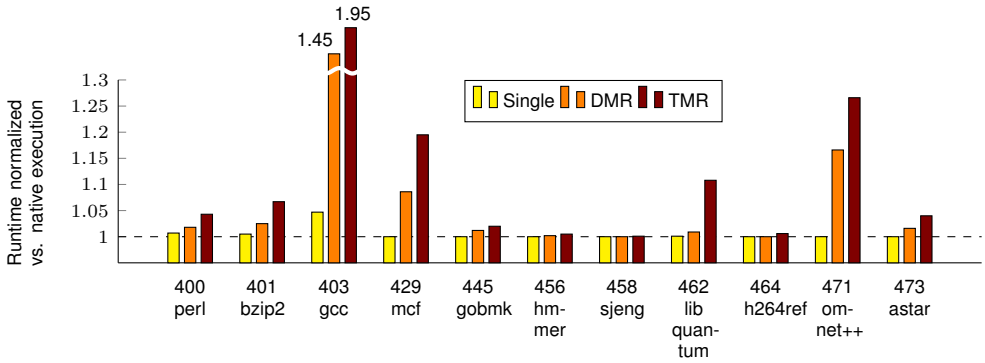

Figure 4: Overhead for replicating the SPEC INT 2006 benchmarks with one, two, and three replicas compared to native execution. Geometric mean overheads: $GM(DMR) = 0.66\%, GM(TMR) = 2.51\%$

problem can only be fixed once we port *Romain* to a 64-bit architecture that allows to use larger address spaces.

There is one thing we can do to reduce memory management overhead, though: By default, both *Romain* and native L4Re use 4 kB pages to manage address spaces. If an application uses large regions of memory, we will observe a page fault for every of these pages, which leads to several thousands of memory management operations. These operations become increasingly expensive for replicated execution, because with N replicas the *Romain* master has to perform N times the amount of memory allocations and handle N times the amount of page faults.

The x86 architecture allows to handle memory using 4 MB sized *huge pages* [Int13]. This reduces the number of page faults that need to be handled by a factor of 1,024. To use this feature, the respective memory regions need to be aligned to an address that is a multiple of 4 MB. L4Re's memory manager provides a mechanism to request huge pages, but applications need to do so explicitly, which the SPEC INT benchmarks don't. We extended *Romain* to inspect the parameters of replicas' memory management system calls. If the application wishes to allocate a memory region larger than a predefined threshold[3], *Romain* automatically sets the respective flags to allocate memory that can be managed using huge pages.

We used a microbenchmark to validate the results of this optimization. Our application allocates a region sized 800 MB and then sequentially runs over this region reading and writing all pages. Table 1 shows the time required to execute this benchmark natively to *Romain* executing one, two, and three replicas. First, we see that using 4 MB pages reduces the benchmark's runtime by 50% because page fault handling clearly dominates this benchmark.

Second, we see that replicated execution has a much higher overhead than we observed for the SPEC INT benchmarks. This is due to the benchmark's nature: Replicas only execute

[3]We empirically found that 1 MB is a good threshold value for our experiments.

	4 kB Pages	4 MB pages
Native L4Re	0.72 sec	0.38 sec
Romain, 1 replica	0.80 sec	0.38 sec
Romain, 2 replicas	2.23 sec	0.58 sec
Romain, 3 replicas	3.12 sec	0.91 sec

Table 1: Effect of using huge pages for replica memory management

concurrently for short periods before raising the next page fault. The majority of time is then spent managing memory in the master. As explained above, the number of replicas is directly proportional to the amount of memory management work the master needs to perform here and hence TMR's cost is about three times as high than native execution.

We re-ran the whole SPEC INT 2006 benchmark suite again with our memory optimization in place. It turns out that these benchmarks are not dominated by memory management overhead at all and the results were the same as already shown in Figure 4. Only for the `400.perl` benchmark we observed a measurable overhead reduction: TMR overhead was reduced from 4.3% to 2.5% by using huge pages.

4.2 Reducing Replica Cache Misses

Reducing the amount of page faults unfortunately did not reduce the overhead for the `429.mcf`, `462.libquantum`, and `471.omnetpp` benchmarks. Our next attempt to explain their behavior was to profile the benchmarks with the help of hardware performance counters. Table 2 shows the amount of L2 and L3 misses we observed for the three benchmarks.

	`429.mcf`	`462.libquantum`	`471.omnetpp`
DMR: L2 Misses x 1,000	2,600	2,500	270,000
DMR: L3 Misses x 1,000	1,300,000	570	6,900,000
TMR: L2 Misses x 1,000	11,000,000	440,000	35,000,000
TMR: L3 Misses x 1,000	5,200,000	387,000	21,200,000

Table 2: L2 and L3 cache misses for `429.mcf`, `462.libquantum`, and `471.omnetpp`.

We see a manifold increase in L2 and L3 misses for all three benchmarks when moving from two replicas to three replicas. This indicates that these benchmarks are extensively relying on data in the cache. While all replicas execute the same operations on the same data, they do so using dedicated copies of this data and cannot benefit from prefetching effects. Instead, the replicas compete for the limited cache space and this appears to be the reason for their high replication overheads.

In Section 3 we argued that we place replicas on cores on the same CPU socket to reduce the CPU cycles required for sending inter-core events. It turns out that this optimization was pre-mature for the benchmarks in question. While running these benchmarks we see a couple of thousand externalization events handled by the master opposed to millions of cache misses. According to Intel's documentation, an L3 cache miss costs about 60 ns

or 160 CPU cycles on our test machine [Lev09]. We therefore concluded that we should optimize for reducing cache misses instead of minimizing signalling performance.

Our test machine's CPU sockets each have an L3 cache that is shared by all six of the local CPU cores. In order to reduce L3 miss rates, we can make more efficient use of the available L3 caches by distributing replicas across sockets. We therefore adjusted *Romain*'s replica placement to place one replica on the second socket when running in both DMR and TMR modes. This lead to the reduced cache miss rates shown in Table 3.

	`429.mcf`		`462.libquantum`		`471.omnetpp`	
DMR: L2 Misses x 1,000	2,600	+/- 0	2,500	+/- 0	290,000	+11%
DMR: L3 Misses x 1,000	930,000	-29%	323	-43%	5,500,000	-20%
TMR: L2 Misses x 1,000	11,000,000	+/- 0	385,000	-12%	34,900,000	+/- 0
TMR: L3 Misses x 1,000	3,600,000	-30%	8,700	-97%	16,400,000	-22%

Table 3: L2 and L3 cache misses for `429.mcf`, `462.libquantum`, and `471.omnetpp` with one replica running on a different CPU socket

We see a significant decrease in L3 misses across all benchmarks. Additionally, we also observed that these decreases manifest themselves in reduced runtimes when replicating the benchmarks. We compare the improved runtimes to the previously measured results in Figure 5. All benchmarks show lower overheads for DMR and TMR execution. The geometric mean overhead for TMR execution for all SPEC INT benchmarks is now 1.8%.

`429.mcf` and `471.omnetpp` still have non-negligible overheads when running with three replicas. Based on the numbers in Table 3, we attribute these overheads to the remaining L3 misses. Lower overheads would then only be possible with a larger cache.

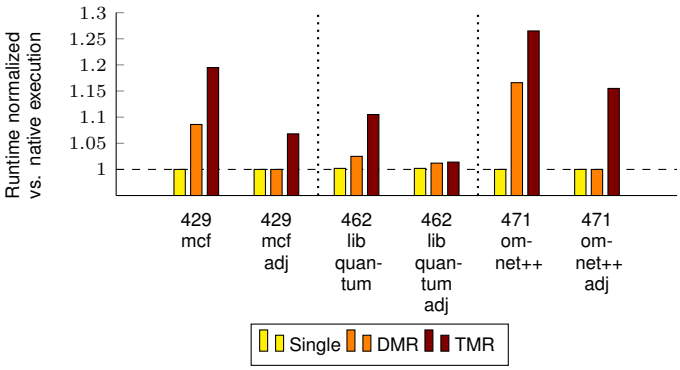

Figure 5: Overhead for replicating the SPEC INT 2006 benchmarks with one, two, and three replicas compared to native execution. Plain benchmark names refer to the previous results (see Fig. 4), benchmark names with *adj* refer to modified core placement.

We conclude from these cache experiences that a replication mechanism needs to be aware of the underlying hardware platform and its specific cache and memory architecture. This observation has already been made in other contexts. The Barrelfish OS [BBD+09] for in-

stance uses a *system knowledge base* to perform efficient message routing between cores. We plan to explore the potential of combining replication with platform- and load balancing in future work.

5 How Much is RCB Hardening?

Our experiments so far showed that *Romain* is able to replicate applications efficiently with an acceptable overhead. Nevertheless, replication only protects user-level code. The system still remains unprotected against hardware errors that occur while executing code within the Reliable Computing Base.

To protect the RCB, we need to apply additional measures. In contrast to user-level code, where our main motivation was to support binary-only applications, we have full control over the RCB's source code. Therefore, applying compiler-based fault tolerance methods may be feasible. Duplicated instructions [RCV+05] as well as redundant arithmetic codes [FSS09] provide such protection without requiring specific hardware support. However, they add additional runtime overhead. We are therefore interested in how this additional overhead will influence replicated execution and ultimately our benchmark results.

Unfortunately, existing compiler-based fault tolerance has only been evaluated using user-level applications and we are not aware of any existing tool that can be used to compile our microkernel. Therefore, instead of performing measurements with a real system, we conduct a thought experiment based on our previous results.

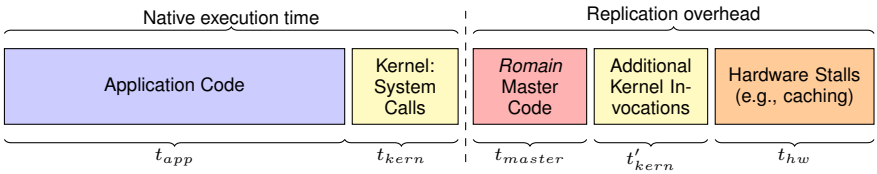

Figure 6: Breakdown of time for replicated execution

To model replicated execution time, we break this time down into its components as shown in Figure 6. Native execution time consists of the time for executing application code t_{app} and the time required for executing system calls in the kernel t_{kern}. When replicating execution, we add time spent in the *Romain* master t_{master}. Furthermore the master may execute additional system calls t'_{kern}. Finally, we suffer from wait times due to unsynchronized replicas, signalling overhead for sending exception messages, and from hardware effects, such as cache thrashing as seen in Section 4, all of which we subsume as t_{hw}.

For our experiment we therefore model native execution time as

$$t_{NAT} := t_{app} + t_{kern} \quad (1)$$

whereas replicated execution time is described by

$$t_{REP} := t_{NAT} + t_{master} + t'_{kern} + t_{hw} \quad (2)$$

If we protect the RCB using a compiler-based mechanism, the kernel as well as the master process will experience additional slowdown and hence their execution times will be multiplied by a factor C depending on the chosen mechanism. The same applies to the kernel part of t_{NAT}. In contrast, the application itself is not modified and therefore does not suffer from additional overhead. Hence, t_{app} remains constant.

We can therefore model the time required for replication based on a protected RCB as

$$t_{PROT} := t_{app} + C \times (t_{kern} + t_{master} + t'_{kern}) + t_{hw} \quad (3)$$

From the results discussed in Section 4 we conclude that applications dominate cache effects, whereas the additional master code and data do not have a significant impact. Even though redundant encoding or additional checking instructions will also add code and data, we still assume this to be negligible in contrast to the application's influence. Therefore, we assume for our model that t_{hw} also remains unchanged with a protected RCB.

We analyzed the SPEC INT 2006 benchmarks to determine how much of their native execution time can be attributed to t_{kern}. We programmed two performance counters to count the `CLK_UNHALTED_REF` event, which counts the unhalted bus cycles. The first counter used the `USER` bit to count cycles spent in the user application. The second counter was configured with the `OS` flag to only count kernel-level cycles. For all of the benchmarks we observed kernel execution time to be less than 0.2% of the whole execution time. To ease modelling, we can therefore safely set $t_{kern} := 0$ and set t_{app} to the native execution times measured in Section 4.

We also set $t_{hw} := 0$ for our analysis, even though we showed that t_{hw} can be a large contributor to replication overhead. Unfortunately, it is hard to precisely determine the ratio of hardware effects to actual additional execution. To get a practical result, we assume the replication overheads measured in Section 4 to be solely software-induced. This will lead to an overestimation of the impact of compiler-based RCB hardening. Nevertheless it allows us to establish an upper bound on the expected overhead.

Our reduced model now becomes

$$t'_{PROT} := t_{app} + C \times (t_{master} + t'_{kern}) \quad (4)$$
$$:= t_{NAT} + C \times t_{REP} \quad (5)$$

To determine the factor C, we consulted an analysis of software-implemented hardware error detection mechanisms done by Schiffel et al. [SSSF10]. The authors compared software-implemented fault tolerance (SWIFT [RCV+05]) and several mechanisms based on arithmetic AN-encoding, such as [FSS09], regarding their runtime overheads and reliability guarantees. They found that SWIFT ($C_{SWIFT} := 1.095$) delivers low overheads, but misses a significant number of hardware errors. In contrast, AN-encoding detects nearly all errors, but has higher runtime overheads ($C_{ANBD} := 3.896$).

We selected the TMR overheads reported in Section 4 and computed the resulting TMR overheads according to our model if we applied either SWIFT or ANBD-encoding to the

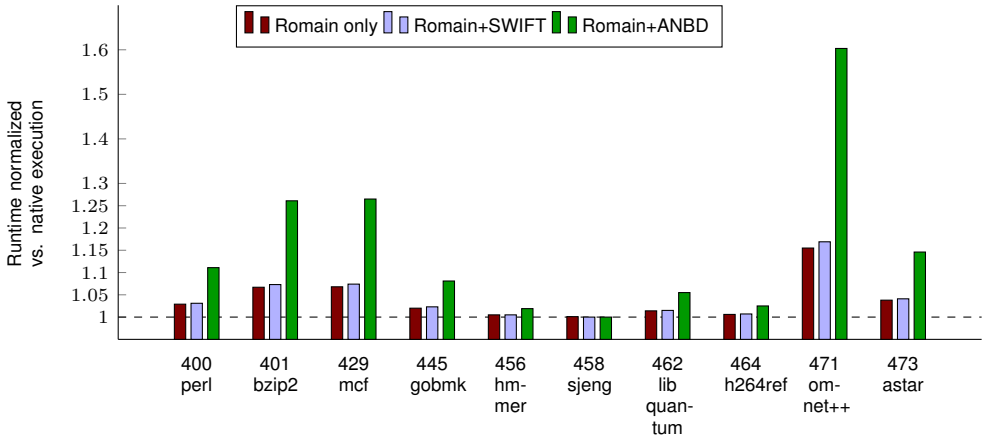

Figure 7: Modelled TMR overhead when protecting RCB execution with software-based fault tolerance mechanisms. Geometric means: $GM(SWIFT) = 1.97\%$, $GM(ANBD) = 7.03\%$

components of our RCB. The results are shown in Figure 7. Applying SWIFT to the RCB adds only tiny amounts of overhead, whereas applying ANBD-encoding would magnify the overheads for those benchmarks that already suffer from high replication overheads when only using *Romain*.

We conclude that the combination of compiler-based fault tolerance and an OS service for replicating binary applications is a promising path for further research as it allows to support a larger range of applications (binary-only) than pure compiler-based methods. Furthermore, the overall system overhead is decreased when combining low-overhead replication with encoding techniques that are selectively applied to those components that cannot be protected by replication. However, we must note that our analysis only considered the performance implications of such a combination. It remains to be researched whether replicated execution can keep up with the reliability guarantees provided by strong arithmetic encoding.

6 Conclusion

This paper analyzed the overheads incurred by the *Romain* replication service when applied to the SPEC INT 2006 benchmark suite. We found that replication cost for these CPU-intensive benchmarks is low with a geometric mean of 1.8% for triple-modular redundant execution. We pointed out that proper placement of replicas on the available CPU cores and the interaction with other services in the system can have a significant impact on the measured overheads and discussed how we avoid those pitfalls in *Romain*.

Given the rising heterogeneity in today's hardware we believe that making system services aware of the underlying platform and finding the right abstractions to let services make platform-aware decisions is an important research goal. Our evaluation shows that reliability mechanisms can also benefit from such knowledge.

Based on our measurements we then developed a model to estimate how combining low-overhead replication with a Reliable Computing Base protected by compiler-based fault tolerance mechanisms would impact our results. Our estimations show that replication can significantly reduce the expected overhead in contrast to applying ANBD-encoding everywhere. It remains to be researched whether this improved performance can be achieved while maintaining strong reliability guarantees.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – `spp1500.itec.kit.edu`). Adam Lackorzynski provided valuable feedback on our measurement setups. Christiane Berndt gave feedback on drafts of this paper.

References

[Aus99] T.M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

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

[DHE12] Björn Döbel, Hermann Härtig, and Michael Engel. Operating System Support for Redundant Multithreading. In *12th International Conference on Embedded Software (EMSOFT)*, 2012.

[ED12] Michael Engel and Björn Döbel. The Reliable Computing Base: A Paradigm for Software-Based Reliability. In *Workshop on Software-Based Methods for Robust Embedded Systems*, 2012.

[FME+12] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[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.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[Int13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325384-046US. March 2013.

[Kri13] Martin Kriegel. Bounding Error Detection Latencies for Replicated Execution. Bachelors thesis, TU Dresden, 2013.

[Lev09] David Levinthal. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ Processors*. 2009.

[LWP10] Adam Lackorzynski, Alexander Warg, and Michael Peter. Generic Virtualization with Virtual Processors. *12th Real-Time Linux Workshop*, 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*, CGO'05, 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.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.

[SSSF10] U. Schiffel, A. Schmitt, M. Süsskraut, and C. Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *Dependability (DEPEND), 2010 Third International Conference on*, pages 51–57, 2010.