Bachelor Thesis

Bounding Error Detection Latencies for Replicated Execution

Martin Kriegel

June 27, 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Björn Döbel

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE BAKKALAUREATSARBEIT

Name des Studenten:	Martin Kriegel
Studiengang:	Informatik
Immatrikulationsnummer:	

Thema:

Bounding Error Detection Latencies For Replicated Execution

Aufgabenstellung:

The Romain replication framework on top of L4Re provides Software-Implemented Redundant Multithreading as a mechanism to andrecover from transient hardware errors. Romain achieves low replication overhead by only comparing states whenever replicas issue a system call or raise a page fault. This replication approach has the drawback that it may take a long time to detect an error if the application is compute-bound and does not issue many system calls. Such a problem can be mitigated by forcing the replicas to raise a fault periodically.

Hardware extensions, such as Intel's Precise Event-Bases Sampling (PEBS), provide a mechanism to trigger periodic faults. The purpose of this thesis is to evaluate the use of such a hardware extension in order to limit Romain's error detection latencies. This includes:

- Design & Implementation of an interface in the Fiasco.OC kernel that gives user applications access to these hardware extensions.
- Application of the new interface to force Romain replicas to periodically raise faults.
- Evaluation of the correlation between replication performance overhead and bounded error detection time.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Döbel
Institut:	Systemarchitektur
Beginn:	15.01.2013
Einzureichen:	15.05.2013

Unterschrift des betreuenden *Hochschullehrers*

[illegible]

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 27. Juni 2013

Martin Kriegel

Danksagung

Ich möchte mich bei allen bedanken die mich während meiner Arbeit unterstützt haben: Allen voran bei meinem Betreuer, der mich wirklich intensiv betreut hat. Angefangen bei Fehlersuche, Hinweisen, Anregungen bishin zu Korrekturen meiner Arbeit, hat mich Björn reichlich unterstützt. Desweiteren möchte ich mich bei Adam und Alex bedanken, welche mir unermüdlich Tips und Lösungen für Probleme bei der kernel-Programmierung gegeben haben. Zum Schluss möchte ich mich noch bei meinen Schwestern, Maria und Eva, für das Probelesen bedanken.

Contents

List of Figures

List of Tables

1 Introduction

Ever since transistor-based computers were invented in the 1960s, it is common that these systems suffer from hardware errors. One particular error is a transient hardware error which can occur out of various reasons: An inherent erroneous processor [2] or errors induced due to different gate switch delays [3], overheating [4], transistor aging, undervolting, frequency scaling [5] and cosmic radiation [6].
Transient hardware errors emerge in various fashions: A transient hardware error may activate a fault that becomes visible as bit flip in the processors registers or in memory. This bit flip causes corruption of data structures or unexpected control flows because of these corrupted data of an application. If these corrupted data is not used anymore or eventually overwritten with valid data, the fault activated by transient hardware errors has no effect on the result or behavior of the application. But, if corrupted data due to bit flips compromises the integrity of an application, then two scenarios are possible [7]: First, the fault can propagate to a failure. A failure is the more fortunate case, because when the application crashes or is able to report an error, we can use general software fault-tolerance mechanism to recover from the fault. The second scenario represents the worst case: A silent-data corruption (SDC) that cannot be distinguished from valid data. Silent-data corruption can lead to a wrong outcome or unexpected behavior due to wrong control flows of an application.
One way of dealing with transient hardware error induced silent-data corruption is replicated execution. The idea of replicated execution is to distribute an application N times over N processors in combination with error detection by state comparison. A faulty replica can then, after the error has been detected, be recovered by a majority of correct replicas. Because it is vital to always have a majority of correct replicas, the error detection period has to be lower than the inter-arrival time of transient hardware errors to avoid multiple faulty replicas. The point where the error detection latency can be reduced to successful recover from transient hardware errors in replicated execution is the main focus of this thesis:
The outline of this thesis starts with Chapter 2 where an introduction is given to fault-tolerance in general, how different errors can be dealt with in hard- and software, leading to the replication approach in software. It follows a brief overview over the target operating system before the structure and function of an existing replication framework is described. Chapter 2 closes by discussing the current error detection latency and its implications for this replication framework.
Chapter 3 is supposed to present the overall goal respectively required error detection latency and how it can be achieved. Then a discussion of three possible approaches are presented and a detailed description of my chosen solution given: An instruction-based watchdog that enables the possibility of frequent surveillance of replicas.
Chapter 4 elaborates the implementation of my chosen solution, discussed in the prior

Chapter. Therefore, an explanation of the hardware features, the operating system support and finally the integration into the existing replication framework is given. It follows the description of the problems I faced during development and how I solved them.
In Chapter 5 a revelation of various experiments and their results is presented. This involves measurements of the overhead that my solution produces in an existing replication framework.
The last Chapter then summarizes my whole work and suggests different improvements and expand-abilities, including the advantages and disadvantages of the chosen approach, in contrast to other approaches.

2 State of the Art

Fault-tolerance is a field for computer-based systems that addresses the accomplishment of an application or service to survive a fault induced due to an error. There are plenty of different error types that can occur in software and as well in hardware. This thesis is focused on the achievement of resilience due to transient hardware errors. One opportunity to detect transient hardware errors in software is redundant execution. In this Chapter a description is given of how the error detection is done in a specific framework developed for redundant execution. In order to lead to this framework it follows an overview of how fault-tolerance is supposed to deal with different error types generally, followed by an introduction to the operating system the framework is built on. Then an explanation of the structure and function of the framework is given, concluding with the focus on the currently deduced error detection latency.

2.1 Fault-tolerance in general

An error can occur in hardware or in software. Depending on how and what part of a system is effected by this error, it has to be decided which fault-tolerance mechanism is appropriate. Fault-tolerance can be reached in hardware or in software: Software fault-tolerance copes with errors which propagate to failures by differentiating between run-time errors and errors caused by poor programming, called software bugs.
Runtime errors are for example *out of memory* or *no space left on device* errors that can be dealt with by for example using *selective retry.* Selective retry presumes that the condition of a resource changes after some time, the operating system for instance kills a temporarily unused process or swaps data to another device. Therefore, it is worth to retry the required function until it either succeeds or a certain threshold of retries is reached.
Software bug induced due to programming faults, in contrast, are much more complicated than run-time errors. They are categorized by the following terms and named after major physicists, respectively mathematicians [8]: A Bohrbug is a deterministic bug that remains constantly no matter how often the system is rebooted. Non-deterministic bugs that fade whenever the system is under surveillance or emerge only occasionally are called Heisenbugs. At last Mandelbugs are bugs whose causes are too complex, that means that they are hard to reproduce and therefore behave pseudo-non-deterministic. There are plenty of mechanisms for dealing with these kind of bugs, one of them being for example the so called *process pairs* [9]: The main process forks a worker process which is responsible for executing the application's code. When this child process crashes, due to a failure, the parent takes over and forks a new worker process that continues from the last checkpoint that was saved on the hard drive. If the fork of a new worker process fails repeatedly, the main process shall *degrade gracefully* and become a worker

process. Usually, solutions like process pairs, selective retry, graceful degradation and check-pointing are used in a combined fashion. Another approach is failure-oblivious computing [10] that prevents a fault to become activated in the first place. In doing so, the application for example ignores writes that are out of the boundaries of a buffer and therefore consequently prevents a buffer overflow.

Errors in software have something in common with hardware errors, but only with persistent hardware errors. Persistent hardware errors either return an error code to the caller, lead to a crash of a service or do not respond at all. The faulty hardware can simply be recovered by replacing it. The only fastidious part is to replace hardware at run-time in order to avoid unwanted temporary unavailability of a service.

Transient hardware errors, in contrast to persistent hardware errors, do not necessarily result in a crash, error reporting or not responding state of an application. But transient hardware errors, on which this thesis is focused on, will rather contaminate the state of the application leading to a misbehavior or miscalculation. Additionally, the faults activated by transient hardware errors occur in a non-deterministic manner and can propagate to failure which appear to behave like a Heisenbug. Other terms for transient hardware errors are bit flips or single-event upsets (SEU) [11].

Transient hardware errors can be coped with in hardware, for example single bit flips in memory can either be detected with parity checks or can further be corrected with special memory, called error-correcting code memory (ECC-memory). It is also possible to detect and correct bit flips in processors by providing additional registers that basically hold redundant data that can be compared to the actual registers for execution. But hard-wired fault-tolerance in processors is expensive and rather a matter of rare purposes like missions in outer space or controlling a nuclear reactor.

For commercial off-the shelf (COTS) products it is more appropriate to handle transient hardware errors in software by using the same approach as in hardware, except that not only data is replicated but rather the whole application. This so called redundant execution is done by replicating the application N times and executing them on different processors. When an application is replicated N times it is possible to correct a replica's contaminated state using the other $N-1$ replicas.

In general it suffices to use triple-modular redundancy (TMR) to achieve resilience concerning transient hardware errors. The error detection is done by comparing the states of all 3 replicas and determining the faulty replica by majority decision. A possible faulty replica can then be recovered by overwriting the memory and processor state, which can be copied from one of the correct replicas. But this implies that TMR is only applicable for errors in a single replica.

Multiple errors in different replicas can be dealt with by increasing the number of replicas. But to detect and recover from multiple errors a majority of correct replicas is needed: Assuming E faulty replicas, a minimum of $N_{min} = (2 \times E + 1)$ replicas is needed for error detection and recovery, with the constraint that the $N_{min} - E$ replicas build a majority of correct replicas. Another way to avoid multiple errors in various replicas is an early error detection. The earlier an error is detected, the higher is the probability to encounter a fault in a single replica before another replica gets contaminated. Increasing the number of replicas on the other hand forces the user to have spare resources and furthermore increases the overhead because of more effort and time spent

in state comparison.
Another approach is to execute the application with double-modular redundancy (DMR), in order to save resources. DMR cannot recover a faulty replica by using a majority decision, but it can detect the error. In combination with frequent checkpoints saved to hard disk or similar devices, both replicas can be rolled back to the last checkpoint after an error has been detected.

2.2 The target operating system

The operating system of interest in this thesis is the L4/Fiasco microkernel [12]. The L4 microkernel was originally developed by Jochen Liedke [13], but there exist various forks of his implementation. One of them is Fiasco.OC which is part of the TU Dresden Research Operating System (TUD:OS). In the following I will give a quick overview over the key features which are significant for this thesis, beginning with the concurrent execution of threads:
Concurrency arises by executing multiple threads on a single resource (CPU). Threads are switched by saving the current CPU state and loading the CPU state of a thread from the ready list to the CPU. This CPU state is called context and therefore the interchange of threads is called context switch. Possible reasons for context switches are traps, interrupts or system calls: Traps are synchronous interrupts, for example preemption due to division by zero or a pagefault.
In L4 traps are presented as exception and delivered to the handler specified on the creation of a thread. But a distinction between a pagefault and other exceptions is made: pagefaults are delivered to a specific pagefault-handler and all other exceptions are delivered to the exception handler. Asynchronous interrupts, in contrast to traps, are caused by entities exterior to the CPU. Timeslice-based scheduling, for example, is realized as an asynchronous interrupt triggered by the clock of the system after the timeslice of a thread expired. Then the scheduler picks the next thread which is present in the ready list (considering priorities) and issues a context switch.
Another feature of Fiasco.OC is that it offers the possibility of programming a software interrupt in the user-land. Software interrupts are treated like asynchronous hardware interrupts.
Fiasco.OC supports *symmetric multiprocessing* (SMP), but all newly created threads are located on CPU 0 per default. To migrate a thread to another CPU, the scheduler provides an interface for defining the required CPU for a specific thread. This enables us to do static or dynamic load balancing in the user-land.
Fiasco.OC is a capability-based microkernel: Capabilities are a security mechanism to ensure authorized access to objects of an application. These objects of an application are therefore encapsulated as long as the application is not willing to offer the capabilities of its objects to other applications (this also applies to threads that execute in the same address space). For invocation of the kernel (system call) on an object, it is inevitable to have the capability of that object. The system call interface of Fiasco.OC passes the object's capability to the kernel and uses inter process communication (IPC) which is supposed to prepare the actual kernel entry. The system call parameters like

the specific system call identifier are passed to the kernel by a thread's personal memory region, the *user-level thread control block* (UTCB). After the kernel finished the system call, the correlated return value is also provided in the UTCB.
The IPC mechanism is incidentally also used for communication between threads. The sender thread puts a message into its UTCB, then calls the IPC *send* system call and passes the capability of the receiver thread to the kernel. If the receiver thread is present and currently in an IPC *receive* state, the kernel will copy the sender's message to the receiver's UTCB. Then the kernel finalizes both send- and receive system calls and might need to switch the receiver and sender thread from a blocked state to a ready state.
Besides for communication, the IPC mechanism is also used for the delivery of exceptions to a thread's exception handler, pagefaults to the thread's pagefault handler and a software interrupt to the interrupt receiver thread that is attached to the software interrupt object.
Furthermore Fiasco.OC provides a mechanism to execute various virtualization technologies [14]. The feature is called *virtual processor* (vCPU) and offers an additional abstraction of privilege modes in the user-land. An operating system usually provides a user-/kernel-mode distinction. With vCPU, the user-mode is further divided into a so called vCPU user mode and a vCPU kernel mode. The application is supposed to operate in the vCPU user mode in its user address space. If the application is preempted due to an exception, pagefault or a software interrupt attached to the vCPU thread, the application migrates into the vCPU kernel address space. After the page-fault or the exception has been handled, the application is resumed and switched back to the vCPU user mode and user address space.

2.3 The L4 replication framework Romain

In order to detect transient hardware errors we need replicated execution as provided by Romain on L4. Romain is built as framework for a transparent replication of binary-only software. The basic model of Romain is depicted in Figure 2.1, which shows that Romain operates in two modes: The first mode is the replicated application executing in the user-land. The second mode is the handler of the application which in the context of Romain is called master and also operates in the user-land. These two modes are possible because of the vCPU feature provided by the kernel. As mentioned above, this feature offers us the opportunity to achieve an user/kernel-mode abstraction when actually operating in user-land. Whenever the application encounters an exception or issues a system call the master is invoked. After the exception was handled the application is resumed. Furthermore, the master decides whether a system call should be executed by itself or by all replicas.

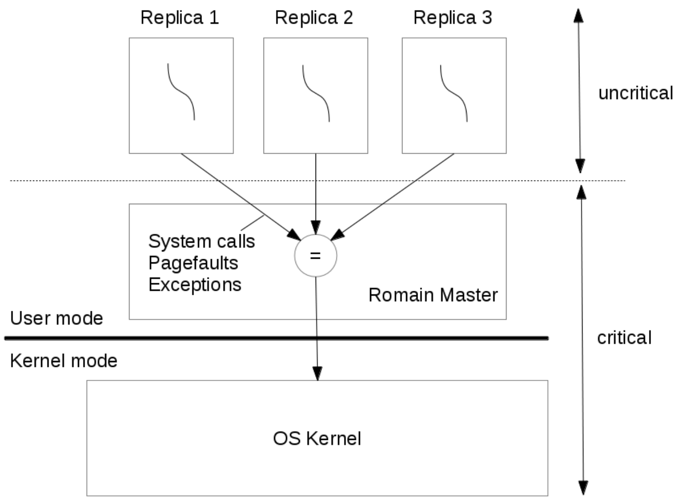

Figure 2.1: Replicated execution model

The framework can only cope with hardware errors that occur while the replicas are running in user-land. When we are in the OS kernel or the master is running, the system is still vulnerable. Hence, the framework only reduces the vulnerability of an application and therefore reduces the probability of a hardware error that induces a failure or miscalculation of the application.

2.4 Error detection latency in Romain

Whenever the application issues an exception or system call, the states of the replicas are compared by the master. Therefore, the rate of checks depends on how exceptions and primary system calls are distributed while the application is running. Figure 2.2

illustrates a possible scenario for one replica. The rectangles represent the execution of one replica, respectively thread T, over the time t in an abstracted periodic manner.

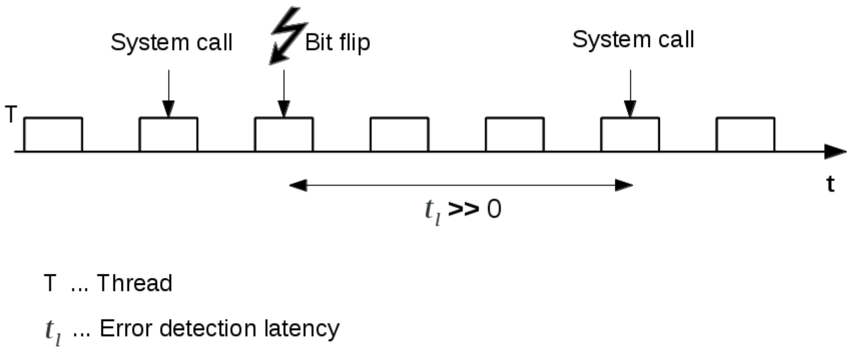

Figure 2.2: Possible error detection latency

This scenario shows that the error detection latency grows with the distance between exceptions or system calls. This has the following drawbacks:

1. *Multiple errors*: When using TMR or redundancy with $N > 3$, it is not possible to recover from multiple errors effecting $N - 1$ replicas. In other words, it has to be guaranteed that a majority of correct replicas are present to recover from multiple errors. As mentioned above, multiple errors in different replicas can be dealt with by increasing the number of replicas. But this leads to more overhead and requires sufficient resources. Another approach is to enhance the strategy of error detection: If we can guarantee that the time between inherent checks is lower than the inter-arrival time of transient hardware errors, we can avoid facing multiple errors in different replicas.

2. *Reiteration*: When using DMR, the recovery is done by rolling back to the last checkpoint. If checkpoints are done rarely, the last checkpoint to roll back to is probably far away in the past. Repeating long paths of execution increases the overall overhead. If the overhead reaches a certain threshold the use of DMR contradicts itself because it was actually chosen to save resources and reduce overhead. More checks, which means more checkpoints, reduce the CPU time required for roll-back, but increase the CPU time for taking checkpoints.

3. *Time requirements*: If the applications has real-time constraints it could be essential that a transient hardware error is corrected until a certain deadline. Forcing periodical checks increases the probability of detecting an error in a prescribed time-frame. It could be argued that error detection and correction at system calls suffices for meeting a deadline in a valid state of the application. This presumes that meeting a deadline means the application delivers an output until a given point in time, respectively issues a system call anyway. But a real-time applica-

tion could have other constraints like for instance certain data has to be written to memory until the deadline.

3 Design

The goal of this thesis is to enhance the replication framework Romain in order to bound the error detection latency. As pointed out in the prior Chapter, the current error detection latency is rather fixed due to the execution model of an application. In this Chapter, a description is given of how the error detection latency can be reduced by firstly illustrating the required outcome. It follows a discussion of possible approaches to achieve this result including an outline of the chosen solution.

3.1 Goal

To reduce the checking interval for an application, one approach is to introduce artificial checks with a period lower than the inherent checking interval due to system calls, exceptions and pagefaults. These artificial checks should lead to a reduced error detection latency as shown in Figure 3.1.

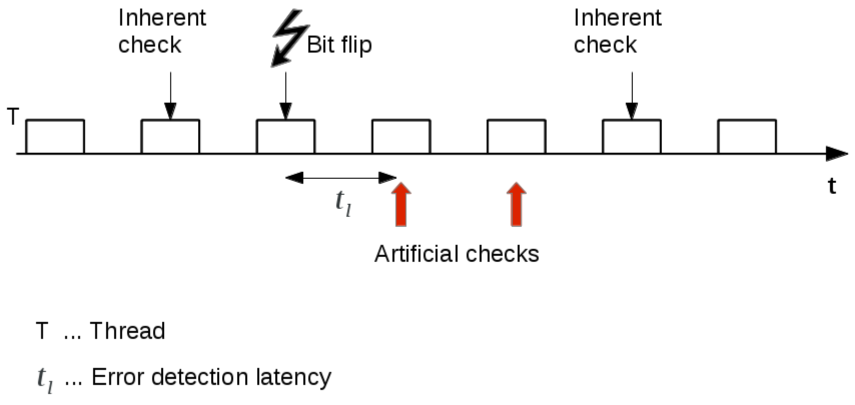

Figure 3.1: Required error detection latency t_l

Finally the user of Romain has to solve the following problem of optimization:

$$\text{Error detection latency} \rightarrow Min. \tag{3.1}$$
$$\text{Induced overhead} \rightarrow Min. \tag{3.2}$$
$$\text{Vulnerability} \rightarrow Min. \tag{3.3}$$

Increasing the amount of artificial checks reduces the error detection latency, but increases the induced overhead and vulnerability due to more time that is spend in the master and in the operating system kernel. The user of Romain would have to find a trade-off between these three optimization tasks. My task is to enable the possibility of solving the optimization problem by the user of Romain in the first place.

3.2 Solution approaches

To achieve frequent checks of the replicas, three solutions are possible:

1. *Alter the application itself.* If we have access to the source code of the application, we can manually add system calls to increase the number of checks. This process is tedious, because we need to modify all branches that an application can take and make sure that the distance between these system calls meet the preferred timing requirements.

2. *Setting debug breakpoints for the application.* Executing a breakpoint instruction leads to a debug trap in the CPU and hence to an interruption of the application. To achieve frequent interruption due to breakpoints, the application's binary needs to be analyzed. But as in the first solution, it is not guaranteed that the application really passes a defined breakpoint. This approach also requires to take care of every branch and hence has a likewise high effort as the first solution.

3. *Using an instruction-based watchdog.* Traditionally a watchdog is used in systems with timing requirements. Therefore, a watchdog only interrupts a process if a certain routine is not triggered by the system itself or from the outside in a given time frame. It can exemplary be seen at the dead man's switch in public transport like trains. If the conductor does not hit the switch until a certain deadline, the algorithm assumes that he has fallen asleep or in the worst case that he is dead. After the timeout the watchdog interrupts the current process and stops the train, to take the train into a secure state.
   The idea of interrupting a process after a timeout can be used to observe replicas in Romain. Instead of a time-based watchdog, which is inappropriate for interrupting replicas at the same point in their execution, a watchdog based on instructions is conceivable. Modern processors provide hardware performance counters that enable us to interrupt an application for instance after exactly N instructions have been executed. We can leverage these features to enforce unmodified replicas to stop after a predefined time without knowing the application's internal structure. The main advantage of this watchdog approach is that it does not suffer from the problems of the first and second solutions. It only needs to be implemented once, the source code of the application is left untouched and it can always be guaranteed that the application is checked after exactly N instructions. I will use the term *timeout* in the following, but actually it is not time but rather the number of instructions the watchdog for Romain is based on.

Adding system calls or setting debug breakpoints could be done automatically before run-time. A source code or binary parser could create a tree of branches that the

application possibly takes. Then, by traversing the tree and counting instructions on each path, a system call could be added or a debug breakpoint could be set on the N-th instruction. The only problematic issue for both approaches are loops: Assuming a loop that is long lasting, but only executes a few instructions in its body. If the system call or the breakpoint is set inside the loop, the performance overhead increases because a check is forced for each iteration. On the other hand, if the system call or breakpoint is set as wrapper before and after the loop, the guarantee of a maximum error detection latency could not be held.
Hence, the more elegant way of enforcing frequent checks is to have an instruction-based watchdog.

3.3 An instruction-based watchdog for L4 threads

In contrast to a timeout-based watchdog we want to measure machine instructions executed by an application. Therefore, hardware performance counters are applicable to measure an application's instructions. For global measurements concerning the whole system it suffices to configure the performance counter once, even in the user-land. As long as no other application or the kernel require to use the same performance counter, everything is fine. But we want a per thread performance counting, to ensure a correct mapping from number of instructions to a thread that really executed these instructions.
Taking into account that the user-land is completely oblivious of context switches, we need the help of the kernel. The kernel has the only ability to ensure that a performance counter is unambiguously mapped to a thread. With the help of hardware performance counters the watchdog's purpose then, is to force a thread to raise an interrupt after it executed exactly N instructions. The interrupt has to be delivered to the exception handler of this thread. This invocation of the exception handler leads, in Romain, to the desired situation that a replica is interrupted and the master can take over.

For the watchdog to work I am assuming:

Assumptions:

- The application is single-threaded, because currently Romain does not support multi-threaded applications.
- The application is treated as black box, respectively as unmodified binary.
- The application is executed on Intel CPUs with Precise Event Based Sampling (PEBS) available. As I will show in Section 4.1.1.2, PEBS is the only possibility of reliably deliver an interrupt after N instructions on Intel CPUs.
- There are only homogeneous CPUs in the system where the application is executed. Otherwise, the instruction count could differ, because on different CPU models some instructions are counted differently.

Furthermore, I describe the required **interface** and functionality of the watchdog with the help of an example: Presuming the user wants to be able to frequently observe its L4 threads after they executed N instructions. Because the user is not aware of context switches and hardware interrupts, he needs kernel support. To fulfill this requirement, the kernel ought to provide an interface to the user-land to mark this thread as thread that uses a performance counter. Because the user has chosen the timeout of N instructions, the enable interface should pass the timeout as parameter to be mapped to the target thread. The user would then expect that the kernel will configure the performance counter to count instructions and preoccupy the counter with the given timeout. The kernel should now take care of the thread's performance counter on context switch and thread migration. If the timeout expired and the kernel receives an interrupt, the kernel additionally should take care of resetting the timeout for the current thread. Now we have enabled a valid performance counting per thread, but it needs more to become a watchdog: We can interrupt the target thread, but this interrupt is handled in the kernel, which is completely oblivious of the requirements of the user-land. On the other hand, the user-land is completely oblivious of hardware interrupts that are handled in the kernel. What we need is a delivery of the hardware interrupt to the user-land, and therefore we can use the opportunity of a software interrupt in L4. The idea is to create a software interrupt in the user-land and pass it to the kernel so that it can be bound to the target thread. On a performance counter interrupt the kernel would forward this interrupt by simply triggering the software interrupt of the user. To summarize the requirements of the watchdog enable interface, the interface needs to provide arguments for passing the capability of the target thread, the capability of the software interrupt and the timeout of the watchdog.
The user can actually decide whether he wants to be only notified that the timeout has been reached or that the target thread is interrupted after the timeout. For notification the only thing the user needs to do is to create a thread which is supposed to receive an interrupt IPC when the software interrupt is triggered. For target thread interruption the user needs to create its target thread as vCPU thread. Therefore, the user can profit from the mechanism that an application is interrupted whenever an exception or pagefault is encountered or a software interrupt bound to the application is triggered. As mentioned above, the application is operating in vCPU user mode and the exception handler in vCPU kernel mode. Because the user requires to count instructions only executed by the application, the kernel needs to make sure that the performance counting is stopped when the thread is in vCPU kernel mode.
After the user has enabled the watchdog for his target threads, a certain need for reconfiguration could emerge: The user could decide to adjust the timeout or to reset the watchdog for a running thread. Furthermore, the user could wish to stop the watchdog for a certain period of time. This implies the need of providing the ability to restart the watchdog as well. At last the user could be willing to use the watchdog only partially for performance reasons. For a partial usage of the watchdog, there needs to be the possibility to disable the watchdog. It would be uncomfortable to use the same interface for managing the watchdog as for enabling the watchdog, because the software interrupt would have to be passed as argument every time. The more appropriate way is to provide a second interface for managing the watchdog for a specific thread.

4 Implementation

The implementation of a watchdog for Romain needs the full facet of features from low-level hardware support, to the operating system kernel to high-level programming mechanism. Starting with the foundation for a watchdog, it is explained how hardware performance counters are configured. Next, it follows a description of the watchdog kernel interface that meets the requirements mentioned in the previous Chapter and how the kernel is supposed to handle timeout-driven interruption of threads with the help of hardware performance counters. Then a outline is given on how all comes together in the replication framework Romain.

4.1 Watchdog support for L4 threads

My goal is to implement a watchdog mechanism that enables us to interrupt a thread after a certain timeout. To achieve a periodic interruption of a thread we need a hardware performance counter which supports interrupts. Intel x86 CPUs offer the possibility to count an event and trigger a hardware interrupt after the counter has overflown. In the following, I describe how performance counters work on Intel CPUs and how they can be used to implement a watchdog.

4.1.1 Performance counter on Intel CPUs

Intel CPUs allow us to count events in all privilege levels and to deliver an interrupt on counter overflow. First, I introduce the standard way of a performance counter setup. It follows, as a second possibility, an extended performance counter feature that I also implemented.

4.1.1.1 Basic setup

Performance counter in Intel CPUs are configurable via Model Specific Registers (MSRs) [1] (pp. 2799), as shown in Figure 4.1.

The essential fields and flags for my implementation, which also represent a rather minimum performance counter setup, are listed in the following:

- **Event Select**: Select an event for a specific architecture.
- **UMASK (Unit Mask)**: Specify more precisely which sub-event should be counted.
- **USR (user mode)**: CPU counts in privilege levels 1, 2 and 3.
- **OS (operating system mode)**: CPU counts in privilege level 0.

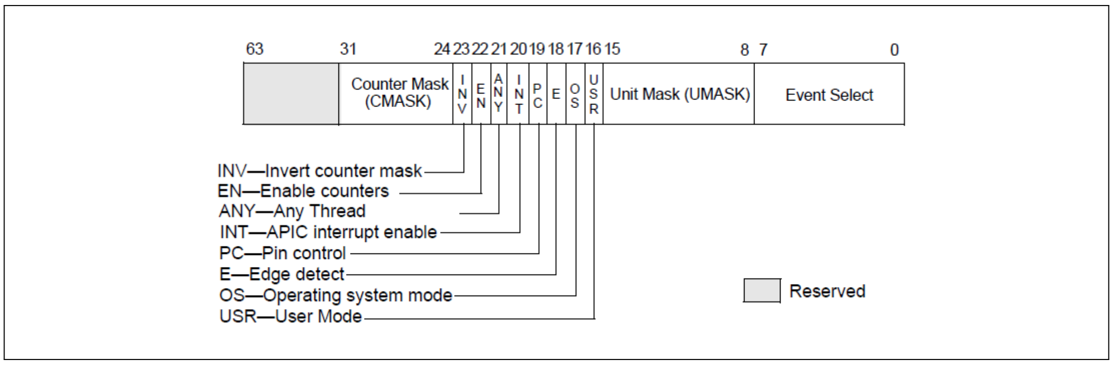

Figure 4.1: Event selection MSR ([1], page 2227)

- **INT (interrupt)**: Enable/disable interrupts on counter overflow by the local APIC.
- **ANY (any thread)**: Count either all hyper-threads on a core or only one thread.
- **EN (enable counter)**: Start/stop the counter.

The basic setup of a performance counter with **Event Select**, **Unit Mask**, **user mode**, **operating system mode**, **edge detect** and **enable counter** was already implemented in Fiasco. I extended this existing interface with the possibility to set the **interrupt** flag and to stop the counter. The Counter Mask, Invert counter mask, Pin control and Edge are zero by default.
The actual performance counter is also located in an MSR and with Intel's performance counter version 3 the length of the counter is 48 Bit. To achieve a watchdog behavior all that needs to be done is to configure a performance counter for a specific event with interrupts on overflow enabled and preset the counter. To interrupt after N instructions, for example, we would preset the counter with the value $-(N)$ in order to reach 0 after N events were counted.

4.1.1.2 Precise Event Based Sampling (PEBS)

The interrupt on counter overflow turned out to be imprecise. This means that there is a variable latency between counter overflow and the performance counter interrupt. In Section 18.8.4.4 of Intel's Developer Manual [1] the problem is described as follows:

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable "skids "in instruction address space.

It is further suggested to use the Precise Distribution of Instruction Retired (PDIR) facility to mitigate these “skids “. But my experiments resulted in the manner that PDIR not really solves the problem. Another disadvantage of PDIR is that it is only available in one performance counter register and if activated, all other performance counters would be quiesced.
Intel developed an extended possibility of performance counting for a bounded amount of events called Precise Event Based Sampling (PEBS). The idea is to save the CPU state exactly when the counter overflows into memory instead of interrupting by the APIC.
PEBS works by defining a debug store area and write the pointer to it in the debug store MSR. This debug store area holds a pointer to the buffer where the CPU state samples should be saved to and an index pointer which points to the last sampled record. Figure 4.2 shows how a PEBS buffer is defined per CPU.

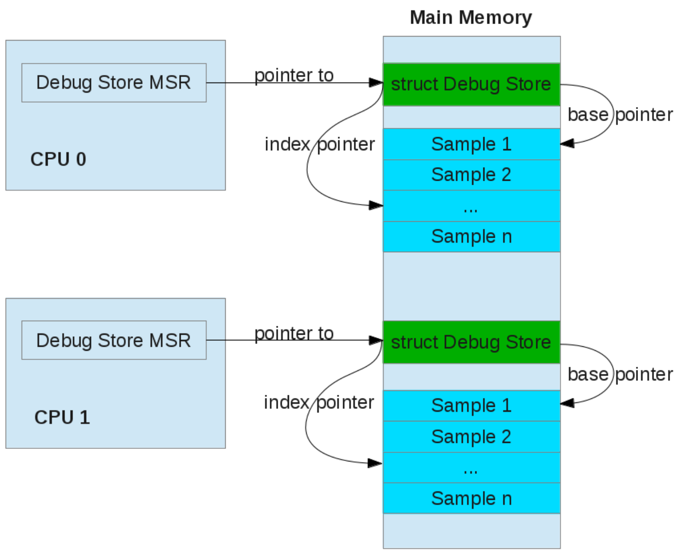

Figure 4.2: Each CPU needs its own buffer to sample to

I implemented the following steps to enable PEBS ([1], page 2238) in the kernel:

1. Check if PEBS is available on this CPU in the misc MSR.

2. Stop performance counter in the global control MSR.

3. Disable PEBS in the PEBS MSR.

4. If the global status MSR indicates that the buffer is overflown: Clear the overflow flag in the global control MSR.

5. Setup a PEBS capable event in the event selection MSR with interrupts disabled on overflow and start it locally.

6. Allocate the PEBS buffer in memory. Configure the debug store area to point to this buffer and additionally determine how many samples fit into the buffer and when the buffer overflow interrupt should trigger. At last the pointer to the debug store area needs to be written to the debug store area MSR.
7. Enable PEBS in the PEBS MSR.
8. Start performance counter in the global control MSR.

Unfortunately, my experiments showed that PEBS also lacks precision. In Intel's Performance Analysis Guide [15] on page 19 the following explanation is given:

The PEBS mechanism is armed by the overflow of the counter. There is a short propagation delay between the counter overflow and when PEBS is ready to capture the next event. This shadow makes the use of the precise event inappropriate for basic block execution counting.

In conclusion, the replication framework Romain has to cope with replicas with different instruction pointers, no matter which performance counter feature is used. In the following I describe the implied problems and how to resolve them, especially how to synchronize replicas in Section 4.2.2.

4.1.2 The kernel interface

In order to fulfill the demands on a watchdog for L4 threads, I implemented two kernel interfaces. One interface is destined to enable the watchdog for a specific thread with a certain timeout and a software interrupt object to get either notified or interrupted. The second interface is supposed to be used for managing the watchdog, which includes a reconfiguration of the timeout, respectively resetting the watchdog, stopping, starting or wholly disabling the watchdog.

4.1.2.1 The watchdog enable system call

```
l4_cap_idx_t target_thread, handler_thread, irq;

l4_irq_attach(irq, label, handler_thread);

l4_utcb_mr()->mr[3] = -(2000);    // initial timeout

l4_thread_watchdog_enable(target_thread, irq);
```

Listing 4.1: Enable the watchdog in L4Re

The **l4__thread__watchdog__enable** method is a wrapper for the actual system call that invokes the kernel. The system call identifier and the software interrupt object are passed to the kernel in the message register (`mr[x]`) using the fields $\mathtt{x} = 0, 1$ and 2. Then the kernel enables performance counting for the given thread:

At first, the **Thread_perf** flag is set in the state of the target thread. This is needed to determine whether a thread is currently performance counting. Then the corresponding MSR will be configured for the selected event and the start value will be loaded to the performance counter register. If the user wishes to get interrupts on overflow, the next step is to set the interrupt flag in the event selection MSR and to set the performance counter interrupt vector in the vector table of the APIC. Regardless of whether the user wants interrupts, the kernel then binds the given interrupt object from user to the current thread. Finally, the actual counter and its configuration are saved in the thread's context.

4.1.2.2 The watchdog control system call

To manage the watchdog, I implemented a second system call which is designed for resetting. Additionally, the watchdog can also be stopped, restarted and completely disabled:

```
l4_cap_idx_t target_thread;

l4_utcb_mr()->mr[1] = 0x1;    // reset watchdog to value in mr[2]
                    = 0x2;    // start watchdog
                    = 0x3;    // stop watchdog
                    = 0x4;    // disable watchdog

l4_thread_watchdog_control(target_thread);
```

Listing 4.2: Manage the watchdog in L4Re

When the watchdog is disabled via system call or when the thread is killed, the **Thread_perf** flag is deleted from the thread's state. The last performance counting thread in the system functions as garbage collector and deallocates the memory for all CPUs for whom PEBS was setup. To determine the last performance counting thread the global performance counter object holds a reference counter of all performance counting threads in the system. This reference counter is incremented whenever the watchdog is enabled for a thread and is decremented when the watchdog is disabled or the thread is killed.

4.1.3 Interrupt on performance counter overflow

4.1.3.1 NMI vs. maskable interrupt

The local APIC can be configured to deliver the performance counter overflow interrupt as Non-Maskable Interrupt (NMI) or as maskable interrupt. The NMI's advantage is that it interrupts the CPU immediately even if interrupts are disabled. Such a behavior is needed in critical situations and is not necessarily appropriate for performance counting. In fact, it is inadvisable to use NMIs for performance counter interrupts, because the interrupt can trigger at inconvenient moments. For example, the interrupt may hit exactly on the transition from user-land to kernel. Then the thread's instruction pointer is the first instruction of the kernel entry and the thread is already in kernel mode. But

the kernel stack pointer is not yet loaded which leads to inaccessible kernel objects and hence the interrupt cannot be handled.
So I chose to use maskable interrupts for performance counting. Maskable interrupts can only trigger if it is permitted. There are paths in the kernel where interrupts are enabled, for example in the kernel's memory management paths. But it is not guaranteed that our target thread, with a pending interrupt at kernel entry, encounters such a path with an interrupt chance. In Section 4.1.5, I describe my handling of performance counters on context switch which is supposed to solve this issue.

4.1.3.2 The interrupt routine

On interrupt the CPU pushes the instruction pointer and the error code to the stack. In addition, the predefined gate entry pushes the trap number, general purpose registers and segment registers on the stack. This will be delivered as *trap state* to the trap handler which is, in the case of Fiasco.OC, the **handle_slow_trap()** method of the current thread. If the trap number is our performance counter interrupt vector, the performance counter interrupt routine is called. Because the interrupt could have been caused by another thread that just migrated to another CPU (discussed later in Section 4.1.5), the interrupt routine is divided into two parts:

1. Ensure that the interrupt can be re-triggered:
   - If counter or buffer are overflown the overflow status has to be cleared.
   - Reset the PEBS debug store by setting the index pointer back to the base-pointer of the PEBS buffer.
   - Reset the interrupt vector in the local APIC vector table (because it changes on every interrupt).
   - Acknowledge the interrupt at the APIC.
2. If the thread that got the interrupt really caused it:
   - Reset the performance counter register to the start value.
   - Inform the user-land about the counter overflow interrupt by triggering the user-provided interrupt object.

If a thread gets an interrupt and has not caused it, then only part 1 is executed. But this should only be the case if a thread is migrated to another CPU and an interrupt is currently pending for this thread on the old CPU. I show how I resolve the problem of migration with a pending interrupt in Section 4.1.5. But first, to introduce the problem, I describe the behavior of vCPU thread on hardware interrupt in combination with triggering a software interrupt in the next Section.

4.1.4 Performance counter and vCPU

A vCPU thread operates in two modes: In the vCPU user mode or in the vCPU kernel mode. These modes are abstractions for the user-land. Hence, the performance counter would count both modes. But to be suitable for the Romain architecture (described in Section 2.3), we need to be able to count user instructions while ignoring those instructions executed by the replication master. To prevent counting in vCPU kernel mode, I implemented that the counter is stopped when a thread enters the vCPU kernel mode and that the counter is restarted when the thread is resumed in vCPU user mode. The combination of a vCPU thread with hardware- and software-interrupts leads to a slightly different behavior, in contrast to an ordinary L4 thread. The process of this combination is depicted in Figure 4.3.

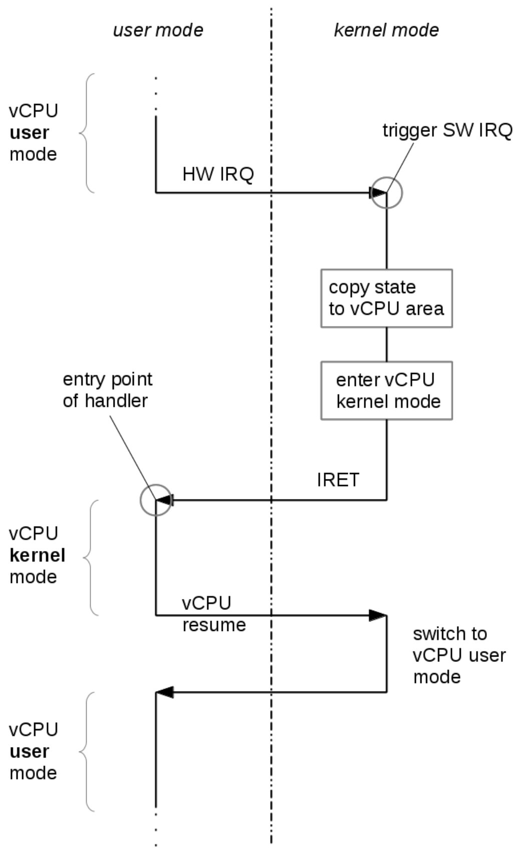

Figure 4.3: Sequence of a vCPU thread with interrupts

Figure 4.3 shows a vCPU thread initially executing in vCPU user mode. When a hardware interrupt hits, the kernel takes over and executes the corresponding interrupt handler. In our watchdog case, the interrupt handler triggers a software interrupt that is attached to the current vCPU thread. Then the kernel prepares the vCPU state and UTCB in order to switch the vCPU thread into the vCPU kernel mode (this is actually an address space migration). On finishing the hardware interrupt, the kernel returns from the interrupt handler with the IRET instruction. The vCPU thread then executes vCPU handler code and eventually issues the vCPU resume system call. To resume the vCPU, the kernel needs to switch the vCPU thread back to the vCPU user mode (migration back to the user address space). Finally, the vCPU thread can keep on executing according to the present vCPU state (possibly modified by the vCPU handler).

The fact that the vCPU thread executes vCPU handler code after the return from the kernel's interrupt handler and eventually resumes into the vCPU user mode, is the main reason for the migration to another CPU problem: If the kernel would permit to be preempted during a vCPU thread migration to another CPU, it would mean to not migrate at all. Because of not continuing with the migration process after the IRET, but instead invoking the vCPU handler, the migration would not be finished. Or even worse, it could leave the kernel's data structures in an inconsistent state. Therefore, preemption during migration is not advisable in our vCPU case and in the next Section I explain how I instead resolve the migration issue.

4.1.5 Context switch and thread migration

Handling performance counter on context switch:

We want to enable a per thread performance counting and we are using one performance counter slot in the CPU. Therefore, we have to use this performance counter slot in a time-division multiplexing manner: Switching the performance counter value and configuration in the CPU at context switch.
A context switch can be done from several points in the kernel. Usually the scheduler is in charge, but the context is also switched if an IPC is sent or the exception handler is invoked. All of them call the **switch_exec_locked()** method of the current context (even the **switch_exec()** method calls **switch_exec_locked()**). I implemented the functionality that is needed for saving and restoring a performance counter in that method. If we are switching **from** a context which has performance counters enabled, the kernel does the following:

- Stop the performance counter
- Save the performance counter value
- Save the event selection MSR

Furthermore, it has to be ensured that no other threads gets an interrupt that is actually dedicated to the current thread. If we entered the kernel it is not guaranteed that we execute a path were maskable interrupts are enabled, which means we could have a pending interrupt that is eventually delivered after we switched the context. To avoid the delivery of pending interrupts to the wrong thread, maskable interrupts are enabled for a short time (two *nop*-instructions) when switching **from** a context with performance counter enabled. If we are switching **to** a context which has performance counters enabled:

- Restore the performance counter
- Restore the event selection MSR

- Start the performance counter

Handling performance counter on thread migration:

When a thread is migrated from one CPU to another we need to save the current performance counter and the event selection MSR as well. Usually, it is sufficient to restore counter and configuration on the new CPU when the thread is rescheduled. I implemented a lazy PEBS setup, so the first thread that is scheduled on a CPU where PEBS is not already enabled, does the setup listed in Section 4.1.1.2 on switching to its context.
It is possible that the performance counter has overflown but we are currently in the kernel and did not encounter a path where interrupts are enabled until this point. If the thread is now migrating to another CPU and the kernel would offer an interrupt chance like in the context switch it could lead to undefined behavior. Especially in the case of vCPU threads an interrupt while migrating could mean to not migrate to another CPU at all, because the exception handler is invoked after the software interrupt is triggered and the migration process is therefor interrupted. But interrupting the migration process could lead to even worse situations, where the thread's state or similar kernel structures are corrupted.
To resolve the interrupt while migrating issue the kernel does not need to enable interrupts, but rather live with threads that get interrupts that are not dedicated to them. On the other hand the kernel has to pretend a performance counter interrupt for the thread that caused it but was migrated. In doing so, the kernel checks whether the performance counter has overflown while the thread is migrating. If necessary the overflow condition, the debug store and the performance counter would be reset. When the thread is rescheduled on the new CPU, the kernel triggers the software interrupt to inform the user-land. The next thread, that is scheduled on the old CPU, which gets the pending interrupt will only make sure that a hardware performance counter interrupt can be re-triggered on this CPU (by resetting the PEBS debug store and acknowledge the interrupt at the local APIC). It does not matter whether the next thread on the old CPU has the watchdog enabled or not, because he is not affected by the interrupt anyway.

4.2 Watchdog integration into the L4 replication framework Romain

To equip Romain with the watchdog feature, I implemented the watchdog enable functionality on replica startup and to additionally reset the watchdog after a certain exception or system call has been handled. Furthermore, I implemented a synchronization mechanism to cope with a possible different progress of the replicas on a watchdog interrupt.

4.2.1 Enabling and managing the watchdog

The watchdog feature is enabled via the configuration file in Romain. On startup the watchdog timeout is set as defined in that file. When the vCPU startup for each replica is issued the interrupt object needed by the watchdog is created. Finally the watchdog enable system call is called with that object.
When running the watchdog is reset whenever the application issues an exception or system call. This is done after the exception is handled and before the replica is resumed.

4.2.2 Synchronizing the replicas

Due to imprecise performance counters, which causes the replicas to interrupt at different instruction pointers, the replicas need to be synchronized. Replicas that are at the same instruction pointer is the premise for the master to be able to compare the replica's states. For replicas which instruction pointers differ, the master is oblivious of which replica made the most progress by only looking at the instruction pointer and CPU state of each replica.
Therefore, I implemented a heuristic synchronization mechanism which assumes that the replica with the largest instruction pointer made the most progress. If that assumption is wrong the synchronization is repeated with another replica with the next largest instruction pointer, until a consensus is reached. In the worst case N synchronization attempts are needed for N replicas. It practically turned out that if the application's memory usage only consists of a few pages, that recovery is in the area of two-digits times faster than to retry until consensus. Hence, I modified the synchronization mechanism to do **early recovery**. For early recovery it suffices to have a majority of correct replicas with equal instruction pointers. The replica that is not synchronized yet would be recovered regardless of whether it is faulty or not. A synchronization retry with the next largest instruction pointer is then only needed, if no majority of correct replicas is present.

Synchronization issues:

Retry and early recovery are the basic mechanisms, but the actual synchronization depends on various situations:

1. **All replicas got the watchdog interrupt**:
   a) *Best case*: All replicas were interrupted at the same instruction pointer. Then the states of all replicas can be compared immediately without any additional synchronization overhead.
   b) *Average case*: A minority of replicas has different instruction pointers. If the majority of replicas has equal states, the minority of replicas can immediately recovered without any synchronization.
   c) *Worst case*: The instruction pointer of a majority of replicas is different. Then, a minimum of one synchronization attempt and a maximum of N synchronization attempts are needed. The synchronization is repeated until

a majority of equal replicas is found, the remaining replicas would then be recovered.

d) *Worst case + Error case*: The even worst case is if no majority of correct replicas is present and additionally one replica is faulty. My implemented strategy then tries to find a majority of correct replicas within N synchronization attempts. If no consensus could be made during these N attempts the synchronization mechanism switches into a **possible error mode**:
During the first N attempts the master marks all replicas which eventually had met the leader's instruction pointer. The replica without a mark is then suspended, because it is probably a faulty replica with a compromised control flow. The other $N-1$ replicas then have further $N-1$ synchronization chances. If a majority of correct replicas can be found, the suspended replica would be recovered and the suspension would be abolished. If no majority of correct replicas could be found, then my synchronization mechanism gives up. This pseudo-graceful degradation situation offers, of course, a wide range of possibilities to deal with: The master could postpone the watchdog to trigger again after a small amount of instructions, which is the most promising solution, especially for the **small-loop** problem. Another possibility is to randomly pick a replica and overwrite all other replicas. But this is a quite dangerous approach, because the randomly picked replica could be a faulty one.

2. **Mixed interrupts**: It can happen that, due to imprecise performance counting, for example one replica encounters another trap like a pagefault, but the other replicas get interrupted by the watchdog. This is the case if the replica with another trap has actually executed more instructions than the other replicas, because some instructions did not lead to an incrementation of the performance counter. The master now assumes that, if the replica with another trap is correct, the other replicas will encounter the same trap after a few instructions.
Therefore, the replicas with the watchdog interrupt are resumed and the watchdog timeout is set to a rather small value. My implementation gives exactly one chance for the watchdog interrupted replicas to catch up in order to encounter the same exception, pagefault or system call like the remaining replicas. If a minimum of one replica misses to catch up, the master switches into the *possible error mode* (see case 1d. *Worst case + Error case*). Then the master determines which interrupt has a majority:

    a) *Majority of watchdog interrupts*: If a majority of replicas with watchdog interrupt could be found, the master marks the remaining replicas with other interrupts as suspended. If x replicas got suspended, the following synchronization of the $(N-x)$ would be issued with an overall $2 \times (N-x)$ number of chances to find a majority of correct replicas.

b) *Majority of other interrupts*: If the majority of replicas, with another interrupt than the watchdog interrupt, are correct the minority of watchdog interrupted replicas can simply recovered.

One replica that encounters another trap can also happen while the replicas get synchronized. This mainly occurs if the replica with the largest instruction pointer is not the replica with the most progress. When the replica with the actual most progress is quasi mistakenly resumed it is possible that this replica encounters another trap. Other interrupts while synchronizing are handled like the interrupts that initially invoked the master:
First, the replicas with watchdog interrupts are resumed in order to eventually encounter the same exception, pagefault or system call like the replica that firstly encounter it. If it is a faulty replica, that encounters another interrupt during actually synchronizing replicas that were interrupted by the watchdog, then the mixed interrupt strategy would cover this situation on the next watchdog interrupt (see case 2. *Mixed interrupts*).

3. **One replica is about to invoke the kernel**: If the largest instruction pointer of one of the replicas indicates to be larger than the address of the kernel entry, the whole synchronization is skipped and all replicas are resumed. A faulty replica that is about to invoke the kernel, is at the latest detected when the other replicas get interrupted by the watchdog. Then the mixed interrupt strategy would become active, described in (see case 2. *Mixed interrupts*).

Unfortunately, small loops are a serious problem for synchronization. There the replicas mostly meet at the same instruction pointer at the first synchronization attempt. But it is not possible to determine which replica made the most iterations in its loop. This means it can not be distinguished between replicas in small loops with different iteration counts and a possible bit flip in one of the replicas. To resolve that I simply assume that at least two replicas are in the same state, then the other one can be recovered even if it is not faulty. If all replicas have different iteration counts in their loops then I give up, reset the watchdog and resume the replicas.

Synchronization modes:

I equipped Romain with the possibility to chose between two methods for synchronization: a **single-step mode** and a **break point mode**. Both modes only define how a replica catches up to another and thence do not implement different synchronization strategies. In the single-step mode the affected replica's executes a single instruction and the following instruction pointer is compared to the instruction pointer of the leader replica. If the replica does not catch up to the leader replica after a certain amount of single-steps (empirically determined by the maximum performance counter inaccuracy) the replica will stop and either a retry is needed or at least two equal replicas were found meanwhile. The single-step mode produces much overhead due to constantly switching between the application and Romain's master after executing only one instruction. Indeed, the single-step mode is only useful for debugging and inaccuracy measurements

of performance counters.
Hence, I implemented a synchronization mode using break points. The replica that needs to catch up to the leader replica simply activates a break point at the leader's instruction pointer. If the affected replica does not hit the break point after a certain amount of instructions the replica would be timed out by the watchdog interrupt (the timeout is empirically determined). Depending on whether at least two correct replicas were found the synchronization has to be repeated and therefore a new break point determined. After every synchronization attempt the used break point needs to be deactivated (restoring the original instruction).

Synchronization primitives:

When the replicas encounter exceptions, pagefaults, system calls and interrupts they migrate to the Romain's master. To compare the states of the replicas, all early arrived replicas need to wait until the last replica enters in the master. This is currently done by a barrier of the pthread (Posix thread) library, based on a mutex (mutual exclusion) and a condition to wait on (from the pthread manpages):

- On entering the master, each replica would try to acquire a lock with the `pthread_mutex_lock(&mutex)` method.
- If the lock acquisition was successful, each replica then calls the `pthread_cond_wait (&condition, &mutex)` method, releasing the lock and waiting for the condition to become true.
- When $(N-1)$ replicas entered the master waiting at the barrier, the last replica (simply determined by counting replicas on enter) would then handle the current event. Eventually the last replica switches the condition to become true by calling `pthread_cond_broadcast(&condition)` and finally releasing the lock with `pthread_mutex_unlock(&mutex)`.
- The remaining $(N-1)$ then try to re-acquire the lock and proceed.

I used the above described barrier to synchronize the replicas on watchdog interrupts: All replicas will firstly pass the existing enter-barrier. If all or at least a majority of replicas got the watchdog interrupt, they will be forwarded to the synchronization methods and eventually arrive at the post-synchronization-barrier. When no consensus can be found, the synchronization is repeated the replicas are forced to arrive at the post-synchronization-barrier again, instead of the usual way of meeting at the enter-barrier.

5 Evaluation

This Chapter is dedicated to show the results of my experiments. The main focus is on how much overhead my implementation produces, measured either with time-stamp counters or with `gettimeofday()` which can be used to determine the past time of an execution. Furthermore, the synchronization behavior is shown in case of measurements concerning Romain. But first, it is described which changes I made on Fiasco.OC and Romain, and which environment I chose to execute my experiments.

5.1 Lines of code (LOC)

I sorted my implemented code to fit into distinguishable subjects and counted the LOC separately. The LOC that I added to the kernel are listed in Table 5.1.

Table 5.1: LOC in the kernel

Subject	New LOC
Performance counter	334
System calls	223
Interrupt handling	135
Context switch	68
Thread migration	31
vCPU	10
Total:	801

The LOC of my user-land implementation is listed in Table 5.2.

Table 5.2: LOC in the user-land

	Subject	New LOC
L4sys:	System call wrapper	36
Watchdog in Romain:	Synchronization	608
	Initialization	129
	Breakpoint mode	59
	Single-step mode	45
	Total:	877

5.2 Environment for measurements

The hardware and software environment that I used for my experiments is described in the following:

Hardware:

All measurements were done on a system with an Intel Core i5-3550 quad-core processor based on the Ivy-bridge architecture. In the following Table 5.3 the specifications of this processor are listed:

Table 5.3: Processor specification

Frequency	3.3 GHz
L1-Cache size	4 x 64 KB
L2-Cache size	4 x 256 KB
L3-Cache size	6 MB
Family	6
Model	58
Stepping	9

The system is furthermore equipped with 4 GB memory that operates at a speed of 1333 MHz.
While I was measuring, I discovered a significant distortion of the results. This distortions are caused by the system management mode, which emulates universal serial bus (USB) keyboard input to PS/2 input. This frequent interference can be solved by disabling legacy USB in the BIOS (Basic Input/Output System), which disables the PS/2 emulation. But the system can still be accessed via the serial terminal, which I then used for my experiments.
Furthermore, I ensured that the Intel Turbo Boost Technology is deactivated.

Software:

I tested and measured with the following versions of the svn (subversion) repositories:

- Fiasco: Revision = 42489 (last changed on 2013-04-14, 23:56:42 +0200 to Revision 42488)
- L4Re: Revision = 42489 (last changed on 2013-03-30, 11:36:55 +0100 to Revision 42397)

5.3 Kernel overhead

To measure the additional overhead introduced in the kernel due to my modifications to all **non**-performance counting threads/applications, I executed the PingPong-benchmark with the original kernel and my modified kernel. The resulting overhead is induced because of if-statements that are checking, whether a thread has performance counter enabled: during context switch (2 if-statements), thread migration (1 if-statement), enter vCPU kernel mode (1 if-statement), resume in vCPU user mode (1 if-statement) and thread kill (1 if-statement). The results are shown in Table 5.4.

Table 5.4: PingPong-benchmark results

Specific benchmark	Original kernel [Cycles]	Modified kernel [Cycles]	Overhead [%]
Short intra IPC			
Shortcut	767 ± 5	851 ± 1	10.94 ± 0.06
No shortcut	769 ± 8	848 ± 1	10.33 ± 0.05
Short inter IPC			
Shortcut	1690 ± 0	1734 ± 1	2.58 ± 0.04
No shortcut	1690 ± 0	1734 ± 1	2.59 ± 0.02
vCPU			
Voluntary switch-to thru kern	431 ± 1	430 ± 1	-0.14 ± 0.19
Intra task (Exc)	1314 ± 1	1335 ± 1	1.58 ± 0.06
Intra task (PF)	1452 ± 2	1450 ± 1	-0.15 ± 0.03
Cross task (Exc)	2322 ± 0	2347 ± 0	1.08 ± 0.00
Cross task (PF)	2569 ± 0	2495 ± 1	1.05 ± 0.03
Cross task (PF+map-sep)	4252 ± 66	4279 ± 65	0.63 ± 1.51
Cross task (PF+map-comb)	3714 ± 3	3715 ± 2	0.03 ± 0.06
Thread switch			
Intra	260 ± 0	286 ± 1	9.89 ± 0.18
Inter	784 ± 0	812 ± 2	3.57 ± 0.21
System call			
invoke_factory_wrong_proto	167 ± 0	174 ± 0	3.95 ± 0.28
invoke_factory_wrong_op	443 ± 0	454 ± 0	2.53 ± 0.11
invoke_inv_cap_timeout0	1008 ± 0	1015 ± 0	0.71 ± 0.04
invoke_empty_cap_timeout0	147 ± 0	148 ± 0	0.54 ± 0.00
Task create (up-only)	12494 ± 9	12468 ± 9	-0.21 ± 0.07
Task create (up+down)	9796506 ± 48	9796420 ± 79	0.00 ± 0.00

The most overhead is induced by the two if-statements at context switch in short execution paths, which can be seen in Table 5.4 in the **IPC**- and **Thread switch**-case. The overhead is reduced in longer paths, like the **vCPU**-case shows, where my context switch and vCPU mode switch modifications are moderated. Overhead results that are negative, which means my modified kernel is faster than the original kernel, is caused by the facts that the benchmark did not pass my modifications, or the values have to be seen with respect to the measurement error: For example, the **vCPU voluntary switch-to thru kern** result is at -0.14% with an error of 0.19%. This can be caused by even one erroneous measurement that distorts the average.

5.4 Micro-benchmarks

I wrote my own micro-benchmark in order to measure the cycles for the watchdog system calls. Furthermore, I measured the cycle count between the hardware interrupt due to the watchdog and the final return from the interrupt routine in the kernel. I use this result to calculate the interrupt overhead in comparison to an execution without interrupts.

Cycle count of watchdog system calls:

With the help of time-stamp counters, I measured the number of cycles that are needed for the watchdog system calls, shown in Table 5.5.

Table 5.5: Cycle measurement of watchdog system calls

System call	Cycles
enable	6,206 ± 29
disable	1,205 ± 2
reset	483 ± 3

Watchdog interrupt overhead:

To measure the watchdog interrupt overhead I created a loop which executes 10 million instructions. These instructions simply add a value to a CPU register and should therefore induce no Cache-/TLB-misses. First, I measured the cycles for the plain loop (with the help of time-stamp counters), without any watchdog interrupts. Second, the watchdog is enabled and the loop is executed with various timeouts (also measuring cycles with time-stamp counters), starting with 1,000 instructions and then doubling the timeout, for each new measurement, until the boundary of 10 million instructions is reached. The cycle count for the plain loop is 10,000,024 ± 7. The cycle counts of the loop with watchdog interrupts and the resulting overhead is shown in Table 5.6.

The resulting overhead curve is depicted in Figure 5.1 with a logarithmic scaled x-axis and a linear scaled y-axis.

The watchdog interrupt overhead shows a approximately proportional curve: halved number of interrupts result in halved overhead. In Table 5.6 can be seen, that one watchdog interrupt has a maximum overhead of 0.57%. The indicated speed-up of 0.13%, due to the 0.22% ± 0.35% average overhead of one interrupt is caused by measurement errors, which distort the average.

Table 5.6: Cycle measurement of watchdog interrupts

Timeout	Number of interrupts	Number of Cycles	Overhead [%]
1,000	12,802	98,537,978 ± 60,234	885.38 ± 0.60
2,000	6,451	55,180,714 ± 180,559	451.81 ± 1.80
4,000	3,236	32,867,127 ± 266,681	228.67 ± 2.67
8,000	1,621	21,682,766 ± 72,856	116.83 ± 0.73
16,000	811	16,079,147 ± 22,739	60.79 ± 0.23
32,000	406	13,072,938 ± 21,374	30.73 ± 0.21
64,000	203	1,169,5249 ± 169,235	16.95 ± 1.69
128,000	101	11,045,151 ± 40,902	10.45 ± 0.41
256,000	50	10,623,874 ± 118,903	6.24 ± 1.19
512,000	25	10,558,720 ± 86,068	5.59 ± 0.86
1,024,000	12	10,196,952 ± 127,227	1.97 ± 1.27
2,048,000	6	10,115,511 ± 121,705	1.15 ± 1.22
4,096,000	3	10,048,891 ± 54,791	0.49 ± 0.55
8,192,000	1	10,021,745 ± 34,968	0.22 ± 0.35

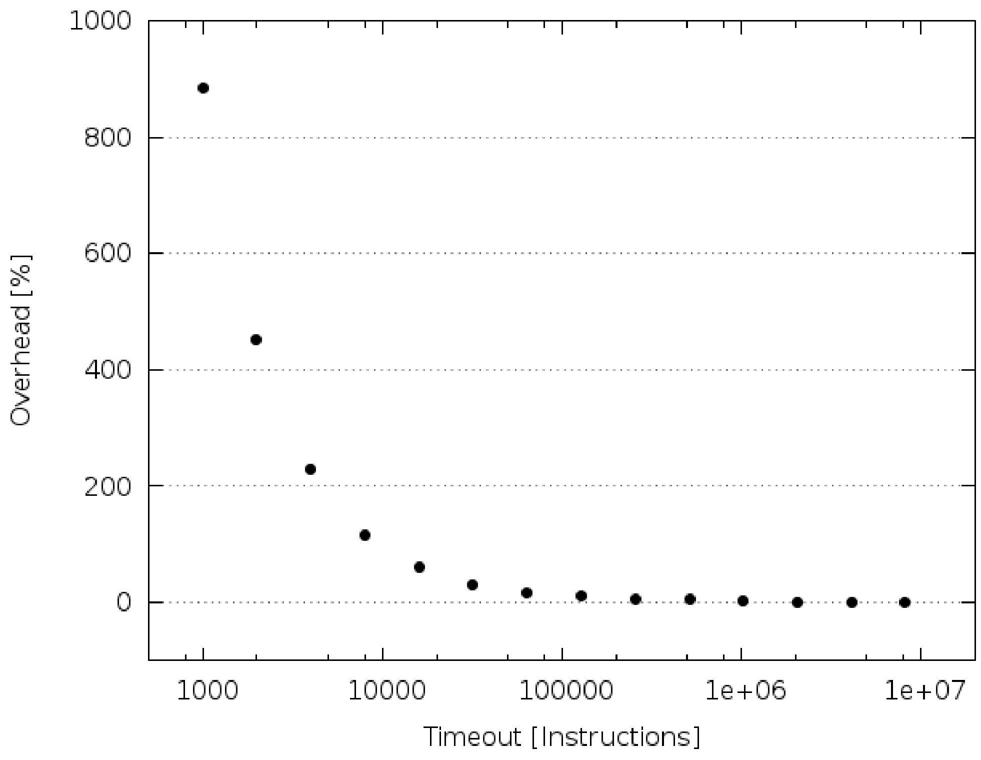

Figure 5.1: Watchdog interrupt overhead depending on the chosen timeout

5.5 Mibench automotive benchmarks

I used the Mibench automotive benchmarks to measure overhead (calculated from execution times) and synchronization behavior of my implementation. Therefore I configured Romain for triple-modular redundancy (TMR). I executed the benchmarks with various watchdog timeouts and to have a reference, I executed the modified kernel and Romain without the watchdog feature. I repeated every following measurement five times, calculated the average and standard deviation of the whole population (marked with $\pm$). Additionally, I measured Cache-misses and Translation Look Aside Buffer (TLB)-misses in kernel- and user-mode, for all following benchmarks. But I only present these misses if it is necessary, respectively in cases of unusual measurement results.
Besides the time and resulting overhead I also present details of the synchronization procedure with the following meaning:

- **Number of interrupts** = total number of watchdog interrupts
- **Number of mixed traps** = number of different events (exceptions vs. watchdog) when entering Romain's master
- **Number of `<x>` sync attempts** = number of synchronization attempts: `x` = 0 means no synchronization was needed, `x` = 1 means one synchronization attempt, and so forth.
- **Number of recovery** = number of stopped synchronizations due to the early recovery approach (a majority of replicas is synchronized)
- **Number of give ups** = number of failed synchronizations

All following benchmarks were executed with a **synchronization timeout** of 400 instructions.

5.5.1 Bitcount-benchmark

The Bitcount-benchmark executes different functions for counting bits in an array:

- Optimized 1 bit/loop counter
- Ratko's mystery algorithm
- Recursive bit count by nybbles
- Non-recursive bit count by nybbles
- Non-recursive bit count by bytes (BW)
- Non-recursive bit count by bytes (AR)
- Shift and count bits

I used the Bitcount-benchmark with the following parameter: `bitcnts <iterations>` with `iterations` = 5,000,000.

In the Bitcount-benchmark, each replica encounters:

- 20 system calls, and
- 12 pagefaults.

The distance between these events over the number of events for a certain distance of one replica, is shown in Figure 5.2:

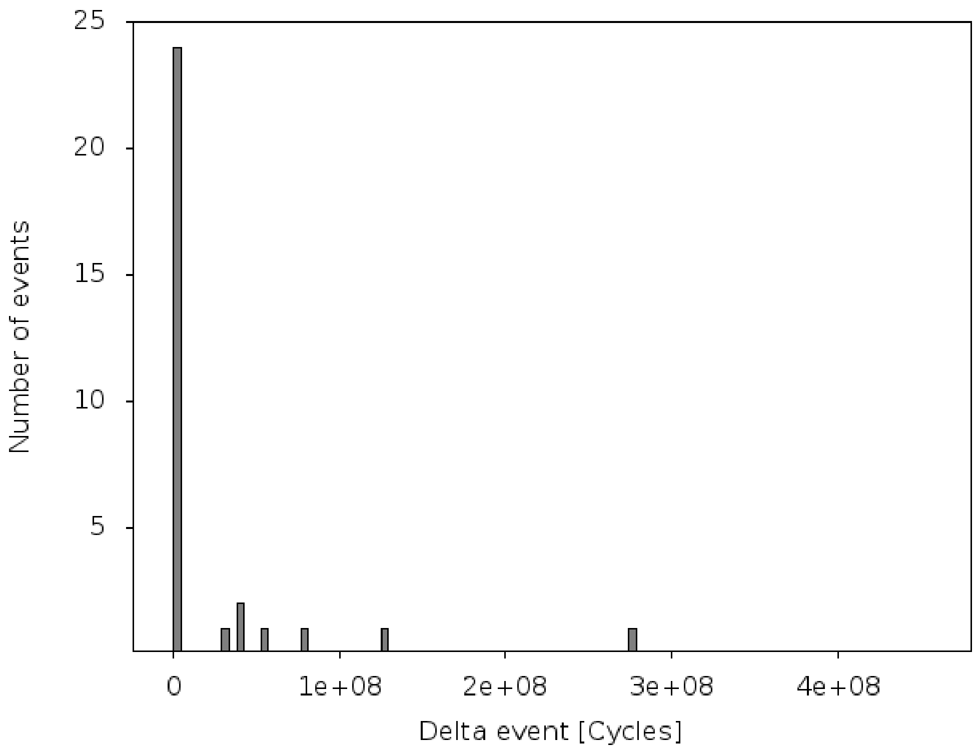

Figure 5.2: Histogram of distances between system calls/pagefaults for the Bitcount-benchmark

The chosen timeout, the correlating watchdog interrupt count, execution time and resulting overhead are shown in Table 5.7. The execution of the benchmark in the original Romain framework took 393.18 ± 0.46 ms.

The resulting overhead curve for the Bitcount-benchmark is depicted in Figure 5.3 with a logarithmic scaled x-axis and a linear scaled y-axis. The synchronization details that occurred during the above measurements, are shown in Table 5.8:

Table 5.7: Time measurements for the Bitcount-benchmark

Timeout	Number of interrupts	Time [ms]	Overhead [%]
10,000	254,795 ± 1	4,458.65 ± 295.30	1,034.01 ± 75.11
20,000	127,624 ± 1	2,136.05 ± 3.75	443.28 ± 0.95
40,000	63,865 ± 0	1,282.64 ± 2.69	226.22 ± 0.69
80,000	31,945 ± 0	1,363.31 ± 84.07	246.74 ± 21.38
160,000	15,975 ± 0	642.51 ± 4.22	63.42 ± 1.07
320,000	7,986 ± 0	518.47 ± 2.89	31.87 ± 0.73
640,000	3,992 ± 0	466.60 ± 2.28	18.68 ± 0.58
1,280,000	1,994 ± 0	420.72 ± 0.58	7.00 ± 0.15
2,560,000	995 ± 0	407.09 ± 0.18	3.54 ± 0.05
5,120,000	496 ± 0	400.15 ± 0.20	1.77 ± 0.05
10,240,000	246 ± 0	396.53 ± 0.30	0.85 ± 0.08
20,480,000	122 ± 0	394.85 ± 0.31	0.43 ± 0.08
40,960,000	58 ± 0	396.56 ± 3.01	0.86 ± 0.77
81,920,000	27 ± 0	393.10 ± 0.47	-0.02 ± 0.12
163,840,000	11 ± 0	393.45 ± 0.71	0.07 ± 0.18
327,680,000	4 ± 0	393.10 ± 0.43	-0.02 ± 0.11
655,360,000	1 ± 0	393.04 ± 0.27	-0.04 ± 0.07

Table 5.8: Replica-synchronization details for the Bitcount-benchmark

Timeout	Number of mixed traps	Number of 0 sync attempts	Number of 1 sync attempts	Number of recovery	Number of give ups
10,000	4 ± 0	101,883 ± 5,266	150,936 ± 5,122	60,338 ± 2,140	1,976 ± 381
20,000	4 ± 0	59,531 ± 1,461	67,254 ± 1,415	26,859 ± 366	839 ± 90
40,000	4 ± 0	23,935 ± 1,370	39,536 ± 1,414	9,826 ± 297	395 ± 54
80,000	4 ± 0	12,413 ± 249	19,338 ± 281	3,682 ± 107	195 ± 84
160,000	4 ± 0	8,497 ± 165	7,424 ± 159	3,314 ± 73	54 ± 11
320,000	4 ± 0	2,338 ± 88	5,624 ± 83	1,240 ± 47	24 ± 10
640,000	4 ± 0	1,832 ± 61	2,124 ± 83	945 ± 38	36 ± 43
1,280,000	4 ± 0	918 ± 103	1,066 ± 101	453 ± 51	9 ± 4
2,560,000	4 ± 0	410 ± 24	578 ± 22	236 ± 10	6 ± 5
5,120,000	4 ± 0	186 ± 10	307 ± 11	122 ± 8	3 ± 3
10,240,000	4 ± 0	79 ± 7	164 ± 5	61 ± 10	3 ± 3
20,480,000	4 ± 0	54 ± 1	66 ± 3	27 ± 3	2 ± 2
40,960,000	4 ± 0	23 ± 7	34 ± 7	14 ± 4	0 ± 0
81,920,000	4 ± 0	11 ± 2	14 ± 2	7 ± 1	2 ± 1
163,840,000	2 ± 0	5 ± 1	6 ± 0	2 ± 1	1 ± 1
327,680,000	1 ± 0	1 ± 1	3 ± 1	1 ± 1	0 ± 0
655,360,000	1 ± 0	0 ± 0	1 ± 0	1 ± 0	0 ± 0

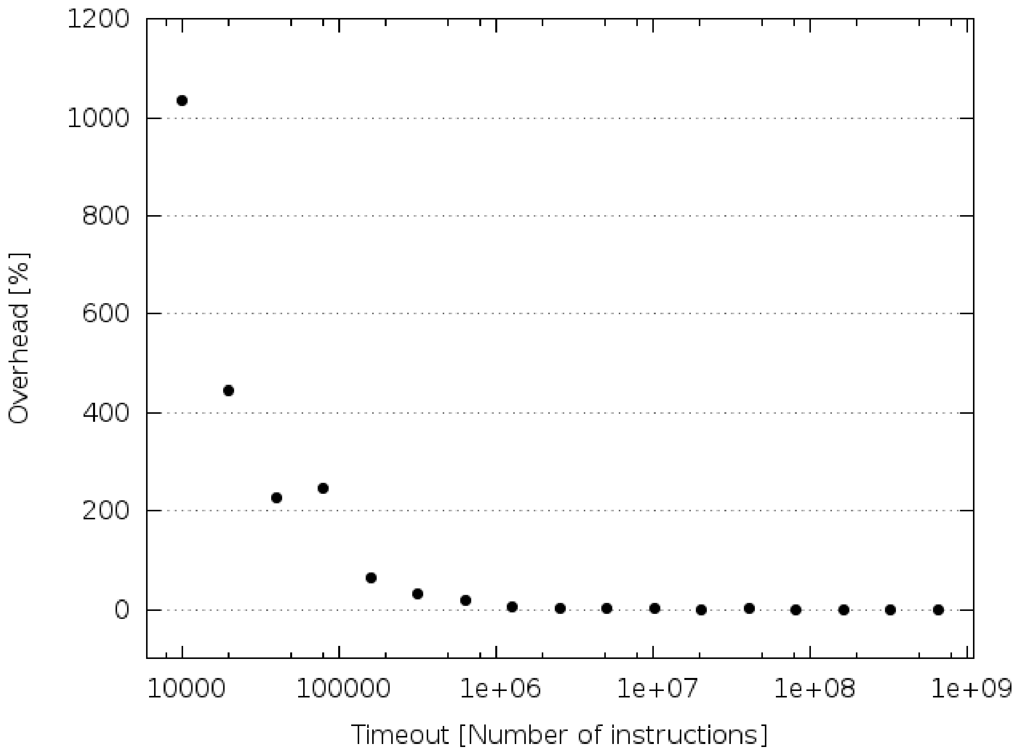

Figure 5.3: Watchdog overhead in Romain for the Bitcount-benchmark

The give up count is in the average between 0.0% and 1.6%, only at the timeout of 81,920,000 and 163,840,000 it reaches 6.7%. The overall overhead curve behaves approximately proportional to the encountered number of watchdog interrupts, except for the timeout = 80,000. To exclude a high number of Cache- or TLB-misses, I further listed the results of my Cache- and TLB-miss measurements for this timeout in Table 5.9.

Table 5.9: Cache- and TLB-Misses for the Bitcount-benchmark

Timeout	LLC-Misses	L2-Misses	L1-Misses	DTLB-Misses	ITLB-Misses
40,000	901 ± 20	276,544 ± 10,740	11,705,248 ± 275,658	839,585 ± 133,714	11992095 ± 54,864
80,000	825 ± 44	133,352 ± 1,796	5,915,934 ± 166,710	412,708 ± 64,889	6,038,728 ± 34,591
160,000	835 ± 34	62,122 ± 1,935	2,912,820 ± 93,388	203,792 ± 33,174	2,955,603 ± 12,602

5.5.2 Susan-benchmark

The Susan-benchmark (Smallest Univalue Segment Assimilating Nucleus) executes several functions on an image, like finding corners. I used the Susan-benchmark with the following parameters: `susan <in.pgm> <out.pgm>` with `in.pgm` = input_large.pgm, and `out.pgm` = output.pgm.

In the Susan-benchmark, each replica encounters:

- 287,239 system calls, and
- 594 pagefaults.

The number of system calls with a distance of ≤ 3000 cycles is 286,757 which is 99.63% of all events. The distance between the remaining 0.37% of events, over the number of events for a certain distance of one replica, is shown in Figure 5.4.

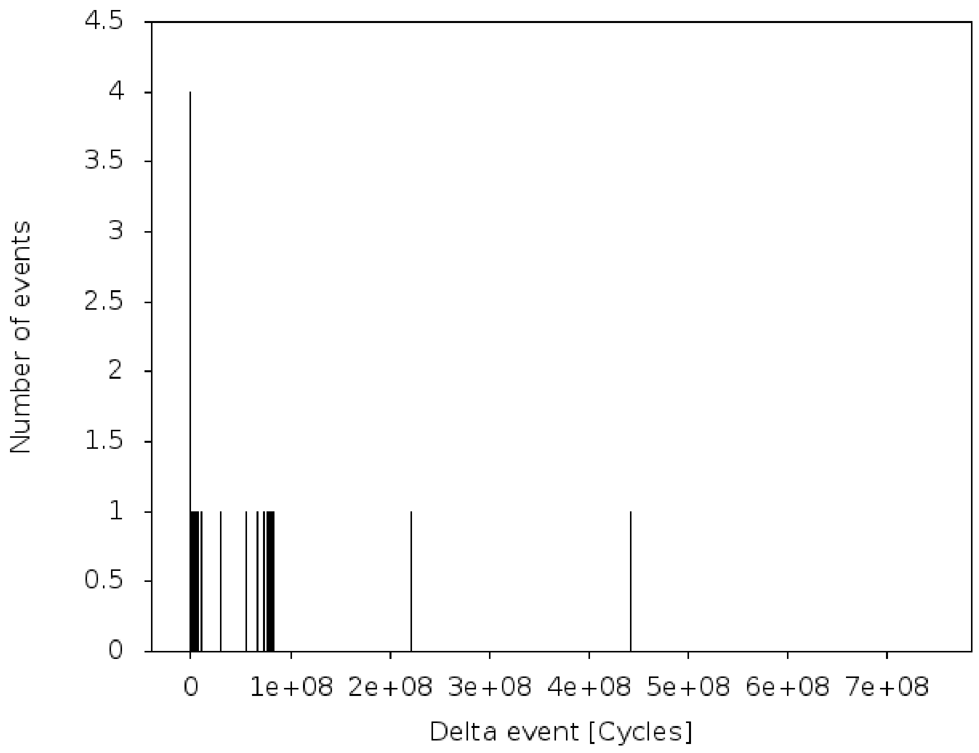

Figure 5.4: Histogram of distances between system calls/pagefaults for the Susan-benchmark

The chosen timeout, the correlating watchdog interrupt count, execution time and resulting overhead are shown in Table 5.10. The execution of the benchmark in the original Romain framework took 3.929 ± 0.292 s.

The resulting overhead curve for the Susan-benchmark is depicted in Figure 5.5 with a logarithmic scaled x-axis and a linear scaled y-axis:

Table 5.10: Time measurements for the Susan-benchmark

Timeout	Number of interrupts	Time [s]	Overhead [%]
100,000	129,087 ± 2	10.905 ± 0.779	177.52 ± 19.82
200,000	64,494 ± 1	7.541 ± 0.087	91.91 ± 2.21
400,000	32,193 ± 1	6.158 ± 0.334	56.72 ± 8.49
800,000	16,051 ± 0	5.149 ± 0.221	31.03 ± 5.61
1,600,000	7,965 ± 0	4.450 ± 0.018	13.26 ± 0.46
3,200,000	3,945 ± 0	4.272 ± 0.011	8.72 ± 0.29
6,400,000	1,933 ± 0	4.176 ± 0.006	6.29 ± 0.14
12,800,000	947 ± 0	4.097 ± 0.003	4.27 ± 0.07
25,600,000	463 ± 0	4.072 ± 0.003	3.62 ± 0.07
51,200,000	226 ± 0	4.105 ± 0.098	4.47 ± 2.49
102,400,000	108 ± 0	4.050 ± 0.003	3.08 ± 0.06
204,800,000	42 ± 0	4.044 ± 0.001	2.92 ± 0.03
409,600,000	20 ± 0	4.044 ± 0.002	2.91 ± 0.06
819,200,000	8 ± 0	4.040 ± 0.002	2.79 ± 0.06
1,638,400,000	2 ± 0	4.043 ± 0.001	2.90 ± 0.02

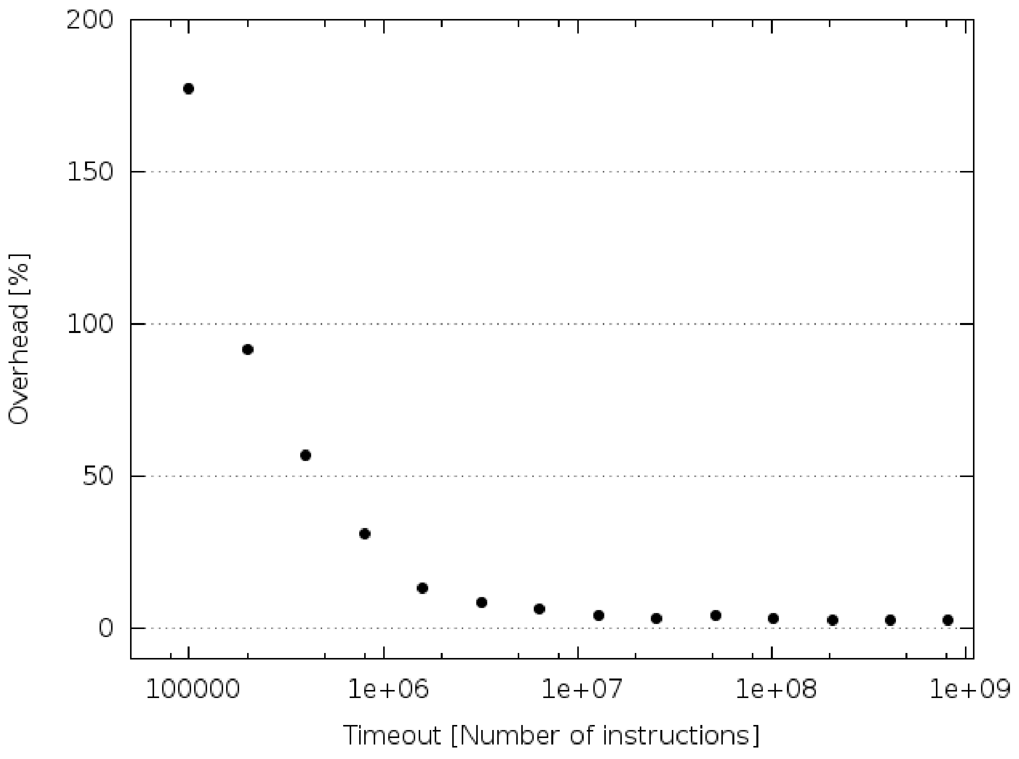

Figure 5.5: Watchdog overhead in Romain for the Susan-benchmark

The synchronization details that occurred during the above measurements, are shown in Table 5.11 and are continued in Table 5.12:

Table 5.11: Replica-synchronization details for the Susan-benchmark

Timeout	Number of mixed traps	Number of recovery	Number of give ups
100,000	111 ± 0	14,593 ± 107	455 ± 61
200,000	108 ± 0	9,478 ± 193	190 ± 58
400,000	102 ± 0	5,062 ± 149	90 ± 8
800,000	97 ± 0	1,897 ± 29	90 ± 16
1,600,000	72 ± 0	1,109 ± 13	17 ± 13
3,200,000	60 ± 0	668 ± 29	6 ± 1
6,400,000	33 ± 0	443 ± 16	3 ± 1
12,800,000	19 ± 0	172 ± 10	2 ± 2
25,600,000	15 ± 0	80 ± 7	3 ± 6
51,200,000	15 ± 1	36 ± 4	0 ± 0
102,400,000	14 ± 0	18 ± 3	3 ± 4
204,800,000	5 ± 0	11 ± 2	2 ± 2
409,600,000	5 ± 0	4 ± 1	0 ± 0
819,200,000	3 ± 0	1 ± 1	0 ± 0
1,638,400,000	1 ± 0	1 ± 0	0 ± 0

The give up count is in the average between 0% and 0.7%, only for the timeouts 102,400,000 and 204,800,000 it reaches 4.8%. The overall watchdog overhead behaves approximately proportional to the number of watchdog interrupts.

Table 5.12: Replica-synchronization details for the Susan-benchmark (continued)

Timeout	Number of 0 attempts	Number of 1 attempts	Number of 2 attempts
100,000	42,960 ± 865	85,616 ± 807	47 ± 4
200,000	22,186 ± 765	42,086 ± 730	29 ± 6
400,000	11,533 ± 306	20,561 ± 305	7 ± 2
800,000	5,455 ± 145	10,500 ± 146	6 ± 1
1,600,000	2,730 ± 48	5,216 ± 56	2 ± 2
3,200,000	1,321 ± 32	2,617 ± 33	1 ± 1
6,400,000	509 ± 27	1,419 ± 28	1 ± 1
12,800,000	280 ± 17	664 ± 17	1 ± 1
25,600,000	138 ± 6	322 ± 9	0 ± 0
51,200,000	62 ± 2	164 ± 3	1 ± 1
102,400,000	30 ± 3	74 ± 5	1 ± 1
204,800,000	9 ± 1	31 ± 3	0 ± 0
409,600,000	4 ± 2	16 ± 2	0 ± 0
819,200,000	2 ± 1	6 ± 1	0 ± 0
1,638,400,000	0.5 ± 0.5	2 ± 1	0 ± 0

5.5.3 Qsort-benchmark

The Qsort-benchmark sorts elements with the Quicksort-Algorithm. I used the Qsort-benchmark with the following parameters: `qsort_small <in.dat>` with `in.dat` = input_large.dat.

In the Qsort-benchmark, each replica encounters:

- 60,016 system calls, and
- 29 pagefaults.

The number of system calls with a distance of $\leq 8,192$ cycles is 60,010 which is 99.94% of all events. The distance between the remaining 0.06% of events, over the number of events for a certain distance of one replica, is shown in Figure 5.6.

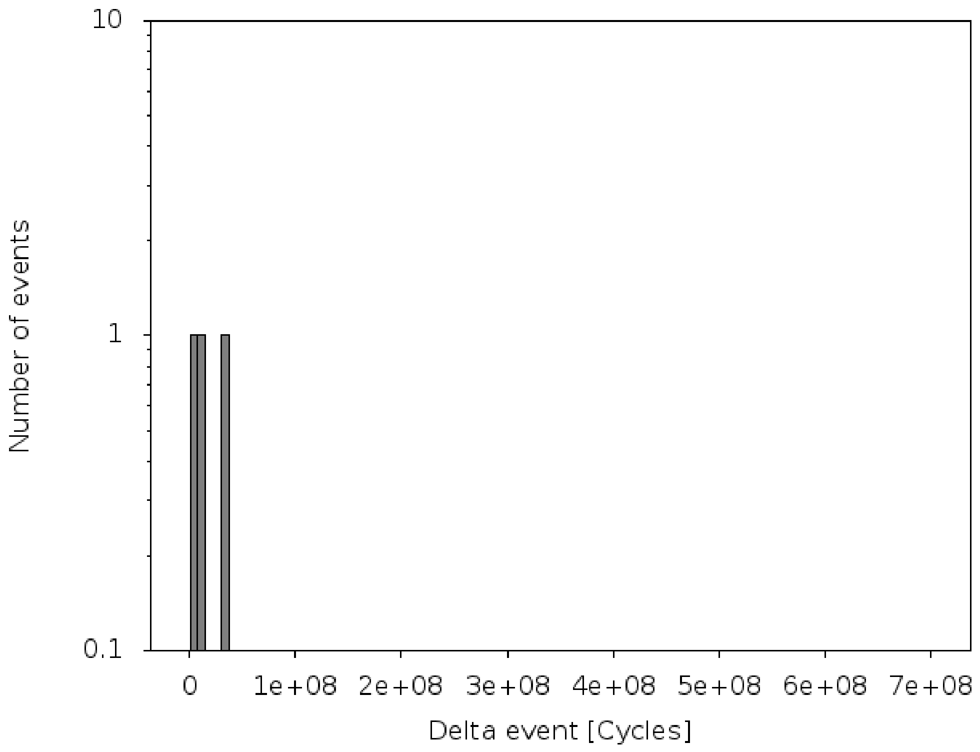

Figure 5.6: Histogram of distances between system calls/pagefaults for the Qsort-benchmark

The chosen timeout, the correlating watchdog interrupt count, execution time and resulting overhead are shown in Table 5.13. The execution of the benchmark in the original Romain framework took 240.26 ± 0.01 s.

The resulting overhead curve for the Qsort-benchmark is depicted in Figure 5.7 with a logarithmic scaled x-axis and a linear scaled y-axis:

The synchronization details that occurred during the above measurements, are shown in Table 5.14 and continued in Table 5.15:

Table 5.13: Time measurements for the Qsort-benchmark

Timeout	Number of interrupts	Time [s]	Overhead [%]
100,000	1,823 ± 1	249.52 ± 0.21	3.855 ± 0.088
200,000	9,115 ± 1	245.22 ± 0.14	2.065 ± 0.056
400,000	4,555 ± 0	242.68 ± 0.01	1.008 ± 0.005
800,000	2,274 ± 0	241.50 ± 0.03	0.516 ± 0.012
1,600,000	1,133 ± 0	241.32 ± 0.07	0.440 ± 0.028
3,200,000	564 ± 0	240.58 ± 0.02	0.133 ± 0.008
6,400,000	280 ± 0	240.44 ± 0.01	0.075 ± 0.005
12,800,000	138 ± 0	240.35 ± 0.01	0.035 ± 0.006
25,600,000	68 ± 0	240.32 ± 0.01	0.023 ± 0.002
51,200,000	34 ± 0	240.29 ± 0.01	0.013 ± 0.002
102,400,000	17 ± 0	240.28 ± 0.01	0.006 ± 0.001
204,800,000	8 ± 0	240.27 ± 0.01	0.003 ± 0.001

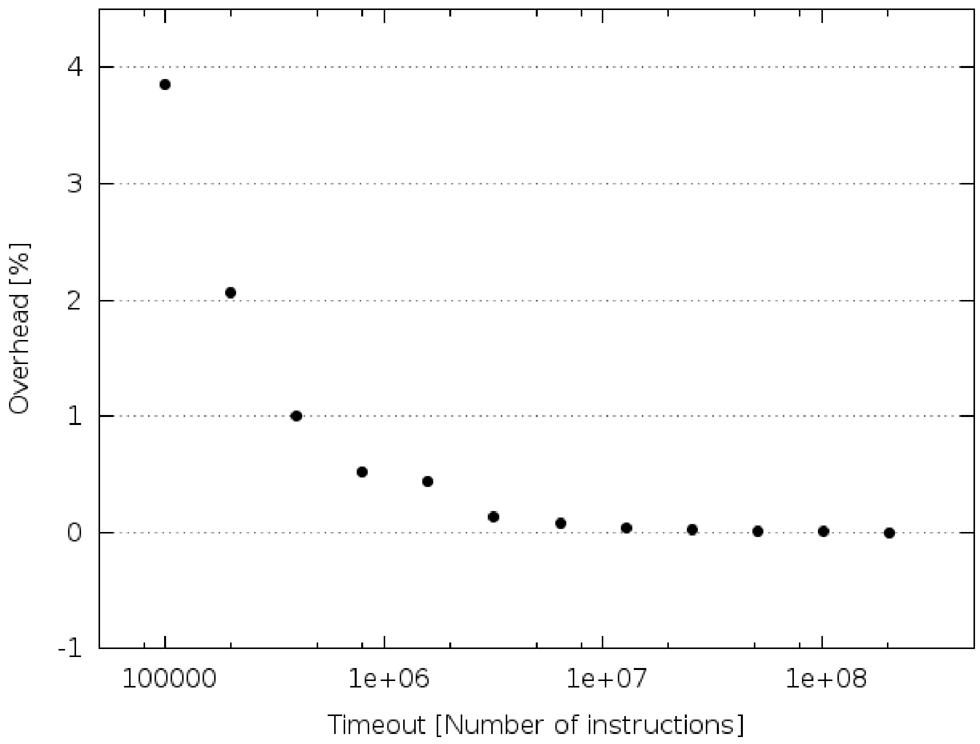

Figure 5.7: Watchdog overhead in Romain for the Qsort-benchmark

Table 5.14: Replica-synchronization details for the Qsort-benchmark

Timeout	Number of interrupts	Number of mixed traps	Number of recovery	Number of give-ups
100,000	1,823 ± 1	7 ± 0	7,310 ± 177	7,884 ± 247
200,000	9,115 ± 1	7 ± 0	3,798 ± 159	3,750 ± 179
400,000	4,555 ± 0	6 ± 0	1,906 ± 9	1,843 ± 30
800,000	2,274 ± 0	6 ± 0	970 ± 19	941 ± 30
1,600,000	1,133 ± 0	5 ± 0	460 ± 9	496 ± 3
3,200,000	564 ± 0	4 ± 0	243 ± 15	232 ± 17
6,400,000	280 ± 0	3 ± 0	135 ± 11	102 ± 15
12,800,000	138 ± 0	2 ± 0	57 ± 11	62 ± 15
25,600,000	68 ± 0	1 ± 0	35 ± 2	24 ± 3
51,200,000	34 ± 0	1 ± 0	16 ± 4	11 ± 3
102,400,000	17 ± 0	1 ± 0	9 ± 2	5 ± 1
204,800,000	8 ± 0	1 ± 0	2 ± 2	5 ± 2

The average give up count is between 33% and 45% and reaches only for the timeout of 204,800,000 a value of 66.7%. The overall watchdog overhead behaves approximately proportional to the number of watchdog interrupts, except for both timeouts with a high give-up rate.

Table 5.15: Replica-synchronization details for the Qsort-benchmark (continued)

Timeout	Number of interrupts	Number of 0 sync attempts	Number of 1 sync attempts	Number of 2 sync attempts
100,000	1,823 ± 1	1,685 ± 44	8,598 ± 203	66 ± 4
200,000	9,115 ± 1	868 ± 8	4,461 ± 169	36 ± 7
400,000	4,555 ± 0	453 ± 14	2,239 ± 22	20 ± 7
800,000	2,274 ± 0	218 ± 12	1,106 ± 31	9 ± 1
1,600,000	1,133 ± 0	93 ± 7	540 ± 3	4 ± 1
3,200,000	564 ± 0	40 ± 2	289 ± 17	2 ± 3
6,400,000	280 ± 0	25 ± 4	152 ± 11	1 ± 1
12,800,000	138 ± 0	9 ± 3	66 ± 11	1 ± 1
25,600,000	68 ± 0	5 ± 2	39 ± 4	0.5 ± 0.5
51,200,000	34 ± 0	2 ± 1	20 ± 4	0.5 ± 0.5
102,400,000	17 ± 0	0.5 ± 0.5	11 ± 1	0 ± 0
204,800,000	8 ± 0	0 ± 0	3 ± 2	0 ± 0

5.5.4 Basic-math-benchmark

The Basic-math-benchmark executes several mathematical functions and operations. I used the `basicmath_small` benchmark in the following.

In the Basic-math-benchmark, each replica encounters:

- 9741 system calls, and
- 8 pagefaults.

The number of system calls with a distance of $\leq$ 13,000 cycles is 9740 which is 99.91% of all events. The distance between the remaining 0.06% of events (pagefaults) is $\leq$ 8,224 cycles, except the first pagefault, which occurs after 30,575,304 cycles.

The chosen timeout, the correlating watchdog interrupt count, execution time and resulting overhead are shown in Table 5.16. The execution of the benchmark in the original Romain framework took 50.56 ± 0.08 s.

Table 5.16: Time measurements for the Basic-math-benchmark

Timeout	Number of interrupts	Time [s]	Overhead [%]
9,000	432 ± 3	50.75 ± 0.37	0.38 ± 0.73
10,000	63 ± 0	50.64 ± 0.04	0.17 ± 0.08
11,000	1 ± 0	50.61 ± 0.01	0.10 ± 0.02

The synchronization details that occurred during the above measurements, are shown in Table 5.17:

Table 5.17: Replica-synchronization details for the Basic-math-benchmark

Timeout	Number of mixed traps	Number of 0 sync attempts	Number of 1 sync attempts	Number of recovery	Number of give ups	Number of passed syncs
9,000	350 ± 8	265 ± 3	157 ± 4	48 ± 2	0 ± 0	6 ± 4
10,000	63 ± 0	22 ± 2	23 ± 16	1 ± 1	0 ± 0	4 ± 3
11,000	2 ± 0	1 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0

The give up count is here at 0.0%, but the passed count reaches up to 6.9%. The timeout-window is in the basicmath-benchmark very small because the watchdog only hits at the time interval between the startup and the first pagefault. Therefore the watchdog is in this benchmark not necessarily needed.

5.6 PEBS vs. Non-precise performance counting

In the following is shown how the watchdog introduces overhead due to the selection of non-precise performance counting instead of using PEBS. Therefore I executed the Bitcount- and the Susan-benchmark with a few timeouts.

Bitcount-benchmark:

The overhead of the Bitcount-benchmark in Romain using non-precise performance counting (in comparison to the execution in the original Romain framework) is shown in Table 5.18

Table 5.18: Time measurements for the Bitcount-benchmark

Timeout	Number of interrupts	Time [s]	Overhead [%]
10,000	253,279 ± 3	4.58 ± 0.06	1,065.42 ± 15.53
20,000	127,248 ± 1	2.17 ± 0.02	452.36 ± 5.13
40,000	63,769 ± 0	1.21 ± 0.01	208.49 ± 2.99
80,000	31,920 ± 0	1.69 ± 0.00	330.19 ± 0.00

The resulting overhead due to the use of non-precise performance counting in comparison to use PEBS is shown in Table 5.19:

Table 5.19: Overhead comparison between PEBS and no-PEBS for the Bitcount-benchmark

Timeout	ΔOverhead [%]
10,000	31.41
20,000	9.08
40,000	-17.73
80,000	83.44

Susan-benchmark:

The overhead of the Susan-benchmark in Romain using non-precise performance counting (in comparison to the execution in the original Romain framework) is shown in Table 5.20

Table 5.20: Time measurements for the Susan-benchmark

Timeout	Number of interrupts	Time [s]	Overhead [%]
100,000	129,014 ± 3	13.75 ± 0.12	249.99 ± 3.02
200,000	64,481 ± 2	9.01 ± 0.46	129.29 ± 11.79
400,000	32,191 ± 1	7.17 ± 0.79	82.45 ± 20.15
800,000	16,052 ± 1	5.27 ± 0.12	34.07 ± 3.07

The resulting overhead due to the use of non-precise performance counting in comparison to use PEBS is shown in Table 5.6:

Table 5.21: Overhead comparison between PEBS and no-PEBS for the Susan-benchmark

Timeout	ΔOverhead [%]
100,000	72.46
200,000	37.38
400,000	25.73
800,000	3.04

6 Conclusion and Outlook

The instruction-based watchdog approach can be used to reduce the error detection latency for replicated execution. But this approach increases not only the overhead caused by the interrupts itself, but rather encounters a high post-interrupt overhead due to the need for synchronization. The developed solution allows plenty of further enhancements, as outlined in the following:
The current implementation of the watchdog is easily portable to other CPU architectures. At the moment my implementation only supports Intel CPUs, but actually every CPU that provides a hardware based performance counting is applicable for the instruction-based watchdog. The fluctuation of the overhead on synchronizing the replicas is mainly induced due to imprecise performance counters. The more precise a CPU can do performance counting, or in other words, the more precise a performance counter interrupt is delivered, the less time is needed for synchronizing the replicas.
Intel's Performance Analysis Guide [15] suggests, to use the *branch instruction retired* event, instead of the *instruction retired* in order to avoid counting errors due to the latency of the PEBS activation. Preempting the replicas after N branches with all replicas at the same instruction pointer would make my synchronization mechanism obsolete. But a branch instruction-based watchdog can only lead to a reduced error detection latency in Romain, in the case of an application that issues periodic branch instructions. An application that resides in deeply nested, long lasting loops and only rarely encounters a branch instruction, would not be suitable for the branch instruction approach.
Romain currently only supports single-threaded applications. The watchdog is also suitable for future enhancements of Romain with multi-threaded application support.
The **early recovery** strategy that I implemented for synchronizing the replicas after a watchdog interrupt is based on the assumption that the application uses only a few pages in memory. If the memory usage of an application increases until a threshold where early recovering is not appropriate anymore, the synchronization strategy should switch to a **late recovery** mode. Late recovery would work by doing the exact opposite of early recovery:
Instead of recovering when at least two correct replicas are present, the recovery would be delayed until all replicas really got the chance to be the leader. The overhead of recovering would have to be exceeded by the overhead of N synchronization attempts for N replicas, before the strategy can be switched to the late recovery mode. The disadvantage of late recovery is that in the worst case, when a transient hardware error occurs and N synchronization attempts are needed for N replicas, the overhead is at maximum. In this worst case, early recovery is always optimal regarding the synchronization overhead.
It remains the **small-loop** problem which either produces far more overhead on syn-

chronization or leads to a higher error detection latency on graceful degradation. The other solution approaches discussed in Chapter 3 are not suffering from this issue. In fact, both solutions do not have the need of synchronizing the replicas at all. But this is the only advantage of these manual approaches.
Furthermore, my implementation and design needs far more experiments to determine its advantages and disadvantages. This includes especially not only test cases and benchmarks, but rather real-life applications to challenge my implemented approach.

Glossary

APIC Advanced programmable interrupt controller

BIOS Basic input/output system

COTS Commercial off-the shelf

DMR Double-modular redundancy

ECC Error-correcting code

IPC Inter process communication

LOC Lines of code

MSR Model specific register

NMI Non-maskable interrupt

PEBS Precise event based sampling

PMU Performance monitoring unit

SDC Silent-data corruption

SEU Single-event upset

SMP Symmetric multiprocessing

SVN Subversion

TMR Triple-modular redundancy

TUD:OS TU Dresden research operating system

USB Universal serial bus

UTCB User-level thread control block

vCPU virtual processor

Bibliography

[1] Intel, “Intel 64 and IA-32 Architectures Software Developer’s Manual,” vol. version 253665, August 2012. IX, 15, 16, 17

[2] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” Micro, IEEE (Volume:25 , Issue: 6). 1

[3] S. Nassif, “The light at the end of the cmos tunnel,” pp. pp. 4–9, Int. Conf. on Application-specific Systems Architectures and Processors, July 2010. 1

[4] D. K. Schroder, “Negative bias temperature instability: What do we understand?,” pp. 841–852, Microelectronics Reliability 47, 2007. 1

[5] M. R. Zhu, D. and D. Mosse, “The effects of energy management on reliability in real-time embedded systems,” pp. pp. 35–40, ICCAD ’04, IEEE Computer Society, 2004. 1

[6] S. Mukherjee, “Architecture design for soft errors,” Morgan Kaufmann Publishers Inc., 2008. 1

[7] A. Avizienis, “Design diversity: An approach to fault tolerance of design fauls,” 1984. 1

[8] M. Grottke and K. S. Trivedi, “A classification of software faults,” 2005. 3

[9] J. F. Bartlett, “A nonstop kernel,” *SIGOPS Oper. Syst. Rev.*, vol. 15, pp. 22–29, Dec. 1981. 3

[10] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., “Enhancing server availability and security through failure-oblivious computing,” in *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation - Volume 6*, USENIX Association, 2004. 4

[11] B. Döbel, H. Härtig, and M. Engel, “Operating system support for redundant multithreading,” 4

[12] O. S. G. TU Dresden, “Fiasco.oc,” http://os.inf.tu-dresden.de/fiasco/. 5

[13] J. Liedtke, “On microkernel construction,” in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, (Copper Mountain Resort, CO), Dec. 1995. 5

[14] A. W. Adam Lackorzynski and M. Peter, “Virtual processors as kernel interface,” 2010. 6

[15] D. D. L. PhD, “Performance analysis guide for intel core i7 processor and intel xeon 5500,” vol. version 1.0, p. 19, 2009. 18, 49