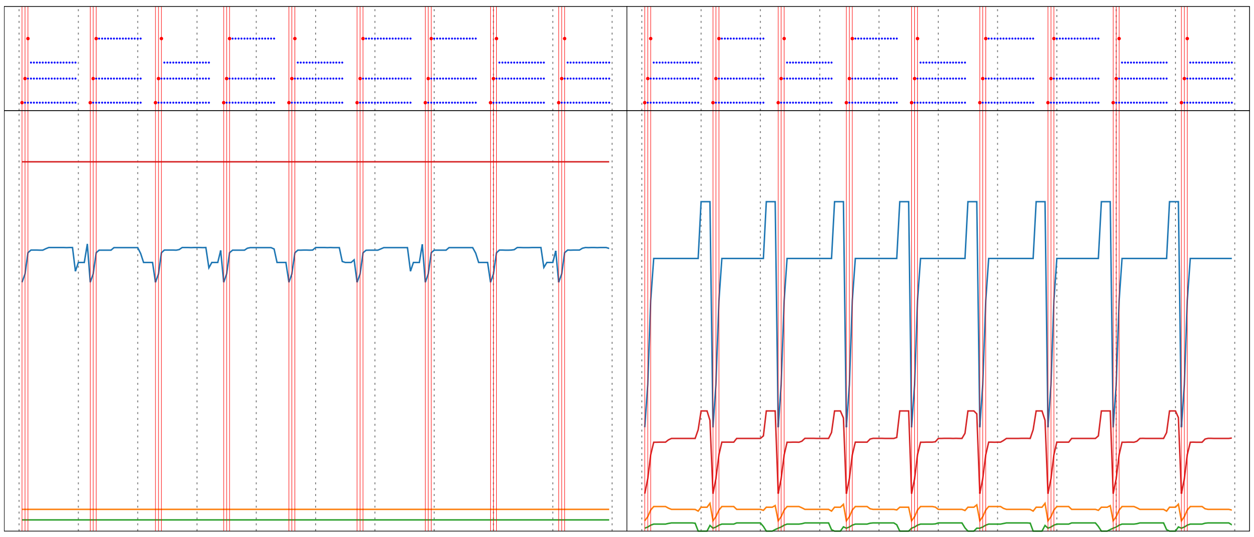

Scheduling mixed-criticality multi-core systems to maximise resource utilisation

Master's dissertation in Computer systems and networks

SIMON HEYWOOD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
and the UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden, 2016

MSc dissertation

Scheduling mixed-criticality multi-core systems to maximise resource utilisation

SIMON HEYWOOD

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
and the University of Gothenburg
Gothenburg, Sweden, 2016

Scheduling mixed-criticality multi-core systems to maximise resource utilisation
SIMON HEYWOOD

Advisor: Daniel Gracia Pérez, Thales
Supervisor: Risat Pathan, Department of Computer Science and Engineering
Examiner: Jan Jonsson, Department of Computer Science and Engineering

MSc dissertation

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement № 610640.

Cover: visualisation of the actions of a worst-case-execution-time controller and quality-of-service strategies on a dual-core processor, as applied to a cyclic statically scheduled program with real-time deadlines (left) and a best-effort program (right).

Typeset in LaTeX
Gothenburg, Sweden, 2016

Scheduling mixed-criticality multi-core systems to maximise resource utilisation
SIMON HEYWOOD
Department of Computer Science and Engineering
Chalmers University of Technology
and the University of Gothenburg

Abstract

Current practice in the safety-critical industry is to run critical application programs on dedicated processors. Adopting widely available multi-core processors to run programs of different levels of criticality in parallel on the same chip would allow the required performance to be achieved with lower energy and material costs. However, the inherent run-time variability would be exacerbated by indirect interference through shared resources, to the extent that no overall benefit would be derived from multi-core technology.

We consider the case of a dual-core ARM processor with a real-time program assigned to one core and a best-effort program assigned to the other. An existing technique from the literature allows us to guarantee safety by stopping the best-effort program – and therefore the interference – whenever a real-time deadline is in danger of being overrun. We show that combining this technique with quality-of-service strategies can result in significant improvements in processor utilisation while maintaining the safety guarantee.

Keywords: mixed criticality, multi-core processor, worst-case execution time, scheduling, quality of service.

Acknowledgements

I would like to express my sincere thanks to Daniel Gracia Pérez, who offered me this research topic and guided my work, for I could not have asked for a more supportive project advisor; to our fellow researchers at ONERA, whose worst-case-execution-time controller this work is based on; to Fentiss, S.L., for supplying the ARM port of Xtratum; to Risat Pathan and Jan Jonsson, for their constructive feedback and advice as academic supervisor and examiner on behalf of Chalmers; to Elad Schiller, for coordinating an excellent master's degree programme, and for his boundless enthusiasm; to my friends, for their inspiration, kindness, and support along the way; and to my mother, for teaching me to learn.

Simon Heywood
Paris, December 2016

Contents

List of figures

List of tables

1 Introduction

Microprocessor manufacturing has seen a significant shift in recent years towards multi-core designs, provoked by diminishing returns in performance from technology scaling, for example the breakdown in Dennard scaling [1] with small feature sizes. Multi-core processors are able to exploit thread-level parallelism and take advantage of shared on-chip resources, allowing the required performance to be achieved with a lower energy footprint.

In a safety-critical system, the failure to meet hard real-time processing deadlines can have differing consequences in terms of severity. The level of severity is classified by the *Design Assurance Level* (DAL) for avionics applications and the *Automotive Safety Integrity Level* (ASIL) for automotive applications. The failure of a system classified as DAL A (the highest integrity level) would have catastrophic consequences. DAL B and C are (to a lesser degree) also safety-critical, while DAL D and E are considered non-safety-critical. An example of a safety-critical application would be a flight management system (DAL B or C), while an example of a non-safety-critical application (DAL E) would be an in-flight entertainment system (DAL E). However, the assigned level ultimately depends on the actual solution as installed in the aircraft.

Current industry practice is to run each safety-critical application program on a single processor. But multi-core technology gives us the opportunity to have applications with different levels of criticality run concurrently on the same processor, thereby maximising resource usage, reducing costs, and minimising energy consumption.

The safety-critical industry has typically guaranteed the real-time deadlines of safety-critical programs by computing the worst-case execution times of the program tasks, and ensuring that tasks are scheduled in such a way as to always meet their deadlines in the worst case. With multi-core processors, this approach is no longer applicable, as the work of Bin [2] (amongst others) demonstrates. This is due to indirect interference between nominally independent programs as a result of their using shared hardware resources. Consequently, there is so much pessimism in the worst-case-execution-time calculations that no benefit can be derived from multi-core technology in such situations.

In order to use multi-core processors in future safety-critical systems, new approaches therefore need to be developed. Bin et al. [3] presented a technique for characterising application behaviour with respect to every potentially shared hardware resource. This technique used stress-testing benchmarks and hardware monitors to measure the extent of the interference between concurrently-running programs as a result of contention for shared resources. Unfortunately, the technique is only suitable for applications where the combination of different programs is limited and controlled, which is not the case in many realistic situations.

For the more general case, Kritikakou et al. [4] addressed the problem of multi-core mixed-criticality by proposing a worst-case-execution-time controller. This controller stops low-criticality tasks (running on other cores) whenever it determines that their continued execution could cause a high-criticality task to fail to meet a deadline. However, this results in all low-criticality tasks being stopped simultaneously, regardless of whether each individual task has an impact on the critical task. Furthermore, the low-criticality tasks are not restarted until after the high-criticality task in question has finished executing, even if restarting them sooner would not endanger the critical task's deadline.

Consequently, the problem that this work aims to solve is that of maximising resource utilisation in a mixed-criticality multi-core system, or – from an application user's perspective – maximising overall performance for the best-effort tasks while still meeting the real-time deadlines of the safety-critical tasks. We refer to this as the quality of service (QoS) provided by the run-time system to the best-effort tasks, and it supplements the deadline guarantees provided by the worst-case-execution-time controller.

2
Background

Influenced by the trend in greater integration of systems of different levels of criticality, along with the move towards multi-core processors, the last decade has seen an abundance of research into mixed-criticality systems. Burns and Davis's comprehensive review [5] reminds us that underpinning this body of research is a fundamental conflict between safety and efficiency: safety is assured through partitioning a system (physically or temporally), while efficiency is maintained through the shared use of resources. The interplay between these two factors becomes increasingly complex as a result of the trade-offs made to maximise utilisation and to minimise energy and manufacturing costs.

Designing a mixed-criticality system with temporal partitioning is possible under certain constraints. For example, Hu et al. [6] consider a pre-emptive event-driven uniprocessor system with fixed task priorities, and show that adaptively shaping the incoming flow of low-criticality tasks at run time can improve system utilisation.

Schneider et al. [7] study schedulability in a mixed criticality system containing real-time tasks with fixed deadlines (high criticality) and control tasks (low criticality) where there is a trade-off between later deadlines and quality of control. They show that, in this context, traditional scheduling is overly conservative, and present an improved schedule synthesis algorithm. However, this work is focused on static scheduling for uniprocessor systems where the behaviour of the low-criticality tasks can be characterised statically.

2.1 A mixed-criticality multi-core system

With a multi-core system, physical partitioning can be used to keep high- and low-criticality tasks on separate cores. In our work, we consider a multi-core processor running a single safety-critical, real-time process (DAL A, B, or C) and one or more non-safety-critical, best-effort processes. The real-time process runs in fixed time slots on a single processor core, according to a predetermined schedule with known worst-case execution time (WCET). The best-effort processes run in the other cores, according to either a cyclic schedule or a fixed-priority schedule, and may be stopped and resumed at any point during their execution.

In a static cyclic schedule (Figure 2.1), the various program tasks repeat periodically, perhaps to sample input data at regular intervals, or to perform a calculation based on that data. This contrasts with a dynamic schedule, where tasks can pre-empt each other at run time. Each real-time task, or sequence of real-time tasks, also has a deadline by which the task or tasks must finish. The deadline is the time by which some action must be taken by the system, for example to operate an electric motor, or to update a display. Even if a task itself does not directly cause an action to be taken, it may be part of a sequence of tasks that results in an action being taken, where each task in the sequence depends on the output of one or more of the preceding tasks.

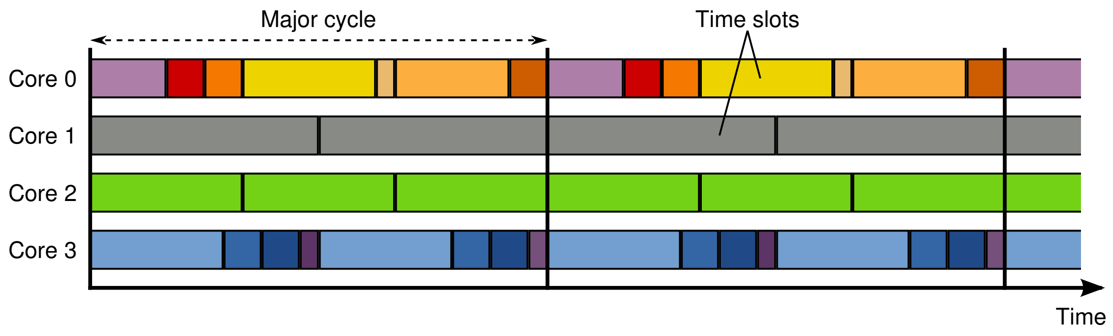

Figure 2.1: Cyclic schedule on a multi-core processor

In addition to the period and deadline, we consider that the worst-case execution time – when running in isolation – is known for each of the real-time tasks, and this information is used to compute a complete static schedule (assuming that a valid schedule is indeed possible). This schedule is implemented as a set of fixed time slots on the processor cores, each time slot containing the sequence of tasks to be executed at that point in the schedule. We refer to the period at which the pattern of time slots repeats itself as the *major cycle*, which is the least common multiple of the task periods.

The method of scheduling the real-time process on a dedicated processor core, in fixed time slots according to known worst-case execution time, avoids the potential for direct interference from the best-effort processes. Direct interference could have occurred if the (potentially sporadic) best-effort tasks were to compete for processing time on the same core as the real-time tasks, and to pre-empt them. With physical separation, the schedule for the real-time tasks can be entirely specified without reference to the best-effort tasks.

However, multi-core processors, such as the dual-core ARM Cortex-A9 shown in Figure 2.2, have a number of resources shared between the processor cores. Although the first-level cache is private to each core, bus access and the rest of the memory hierarchy are shared by all of the cores. Indirect interference occurs, for example, when best-effort tasks write to the same cache line as a real-time task, causing the cached copy to be updated or invalidated. If this happens frequently, the resultant bus traffic and cache operations can cause significant delays to the real-time tasks. Indeed, interference through shared resources is a well-known problem, particularly for shared memory operations [8].

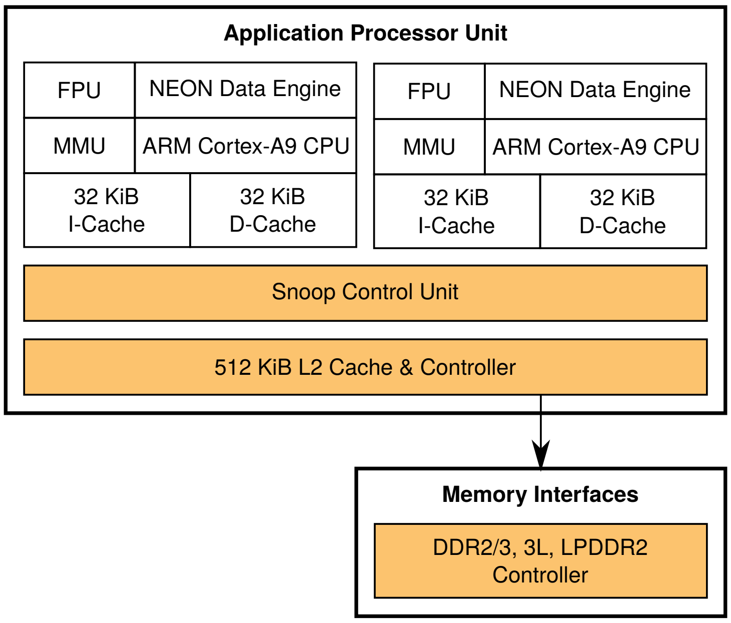

Figure 2.2: Shared resources in a dual-core ARM Cortex-A9 architecture

2.2 Using a run-time system to control WCET

Kritikakou et al. [4] describe a run-time system where regular checkpoints are added to the real-time program in order to detect potential deadline overruns before they happen. Initially, all processes (real-time and best-effort) are run concurrently, but a safety condition (Equation 2.1) is evaluated at each of the checkpoints; if it is not satisfied then the real-time process is in danger of overrunning its deadline, and so the best-effort processes are stopped. The real-time process can then continue running in isolation.

$$RWCET_{iso}(x) + W_{max} + t_{SW} <= D_C - ET(x) \tag{2.1}$$

$RWCET_{iso}(x)$ is the remaining worst-case execution time of the real-time program from checkpoint x, calculated as if it were running in isolation; W_{max} is the maximum worst-case execution time (assuming maximum interference) from checkpoint x to checkpoint $x + 1$; t_{SW} is the overhead involved in stopping the other (best-effort) processes; D_C is the real-time deadline; and $ET(x)$ is the elapsed execution time measured at checkpoint x.

As long as $RWCET_{iso}(x)$, W_{max}, and t_{SW} are known, and that we are capable of measuring $ET(x)$, we can use this mechanism to guarantee that the real-time program will not miss the deadline D_C.

2.3 Nomenclature

In the context of our work, we employ a number of terms with specific meanings in relation to task scheduling, so it will be helpful to provide clear definitions here.

A *slot* is a temporal subdivision of a cyclic schedule on a particular processor core.

The *major cycle* is the period at which the schedule repeats itself across all processor cores.

A *partition* is a logical set of time slots on one or more processor cores to which we map a program.

A *plan* defines a complete schedule (a major cycle), mapping partitions to slots. This is a term used by some hypervisors, which allow the schedule to be changed at run time by switching from one plan to another. In the avionics application domain, different plans might be optimised for the various phases of the flight: take-off, cruising, and landing. Switching plans can also be used for other purposes, for example dynamic reconfiguration after the failure of a processor core so that the system can continue operating.

A *task* is a function (that potentially calls other functions) in the program. It generally carries out an identifiable application task that has similar behaviour across multiple occurrences of it in the same program.

A *job* is an occurrence of a task at a particular point in a given slot. This means that each job repeats once every major cycle. A slot may contain multiple jobs.

2.4 Maximising utilisation

Our work aims to build on the WCET controller technique described by Kritikakou et al. by using it to guarantee the deadlines of critical tasks, while applying new techniques to maximise the overall performance of best-effort tasks.

We define the *utilisation* for slot i as the ratio of execution time ET_i to slot length L_i. The overall utilisation is therefore given by Equation 2.2.

$$\sum_i \frac{ET_i}{L_i} \tag{2.2}$$

It is possible to achieve higher utilisation than that offered by the WCET controller alone because the level of interference can vary with time, and does not necessarily remain constant during a given time slot. This in turn is because the pattern of shared resource (memory, bus) use varies with time: each task will have its own particular characteristics.

Rather than try to profile every combination of tasks running concurrently on the processor, and then fine-tuning the schedule on that basis, our approach is to infer at run time the periods of greatest interference, and to provide a strategy to adapt accordingly. In any case, as well as potentially being prohibitively complex, such profiling may well be meaningless for best-effort programs whose exact behaviour is not known until run time.

2.5 Limitations

We do not consider the case where there are multiple critical process running at the same time, or at least assume – as do Kritikakou et al. – that the problem of determining a valid schedule for multiple critical processes has been solved. The latter is a difficult research problem and is beyond the scope of this work.

3
Methods

In order to validate the theoretical aspects of this work, we set up a dedicated hardware platform that would allow us to make precise measurements and to experiment with techniques for maximising utilisation. In terms of software, we required a real-time hypervisor with configurable time slots, and a run-time library that could schedule arbitrary application and benchmark programs while recording measurements such as the cycle counter, instruction counter, and cache events from the hardware performance monitoring unit.

3.1 Hypervisor

Xtratum is a hypervisor for safety-critical real-time embedded systems, designed in particular with support for aerospace applications (ARINC 653) in mind [9][10]. It offers a low-overhead, low-footprint virtualisation framework based on research from the *Universitat Politècnica de València*, and is developed by a spin-off company, Fentiss, S.L.

Virtualisation is a relatively old technique that has recently started to be used in embedded systems, and consists of modelling enough of the hardware in a hypervisor to allow multiple operating systems (or programs) to be run independently in a partitioned environment.

One of our aims being to measure the performance of – and interference between – the various benchmark and application programs, rather than that of the hypervisor, a more substantial embedded operating system would have been unsuitable. Indeed, we assume that the overhead involved in running the hypervisor is negligible, and does not therefore significantly affect our measurements. It should, however, be possible to estimate the overhead from some of our hardware performance measurements, and thereby test this assumption.

For the purposes of this study, we used the ARM port of Xtratum, which is currently under active development. Through collaboration with Fentiss, S.L., we were able to take advantage of experimental features needed to implement our quality-of-service strategies, notably the ability to instantaneously stop and restart an arbitrary program during a slot for which it is scheduled.

3.2 Hardware platform

For the experimental hardware, we chose the Zynq-7000 ZC706 evaluation kit. The Zynq-7000 system-on-a-chip combines a field-programmable gate array (FPGA) with a dual-core ARM Cortex-A9 (ARMv7) processor, along with a DDR3 SDRAM interface and a number of peripherals [11]. However, for the purpose of this work we did not use the FPGA, since we were only interested in running benchmark programs and our run-time system on the dual-core Cortex-A9 processor.

The main reason for choosing this hardware was that it was supported by Xtratum, with whose developers we were able to collaborate during the course of the project. In addition, the Cortex-A9 has a set of hardware counters for monitoring processor performance. These counters can be programmed to measure a range of activity such as memory access, instruction dispatch, and processor cycles. The counters are programmed and read via a set of ARMv7 coprocessor instructions [12][13].

3.3 Run-time system

We developed a run-time system that runs on top of Xtratum and has three main components: (i) a resource manager, (ii) a fine-grained scheduler, and (iii) a monitoring subsystem.

The *resource manager* is the intelligence in the run-time system. It makes decisions based on the run-time monitoring data, with the objective of maximising utilisation, and provides hints to the best-effort programs so that they can adapt their behaviour to fit in with the real-time programs, and avoid being stopped by the scheduler.

The *fine-grained scheduler* component starts each program task in turn, including the resource management and monitoring tasks, and enforces the decisions made by the resource manager at run time by making adjustments to the schedule for the best-effort programs.

Tasks are run according to a cyclic schedule with a fixed priority assigned to each task. This is equivalent to a pre-determined sequential order. For each slot, the scheduler reads the list of jobs from the current plan, calling each one in turn. (Control returns to the scheduler when a job ends.)

The same scheduler is used for both the real-time program and the best-effort program. The latter, however, is asynchronous with respect to the cyclic schedule: from the scheduler's viewpoint it is a single job that never returns, running continuously across each of its time slots from one major cycle to the next.

The *monitoring subsystem* gathers run-time resource usage data from low-level hardware counters, aggregating it for each job and time slot. On the Cortex-A9 processor, this is done by executing ARMv7 assembly instructions that configure the performance monitoring unit and retrieve its counter values from the coprocessor.

The monitoring functions are called by the run-time scheduler before and after each job; the differences in hardware counter values during that time – an indication of the job performance – are then stored in data structures held in shared memory. Statistical information can then later be retrieved and acted upon by the resource manager.

Figure 3.1 shows the run-time sequence of interactions (function calls) between the fine-grained scheduler (LRS), monitoring subsystem (MON), application programs, and resource manager (LRM) during a major cycle with two slots.

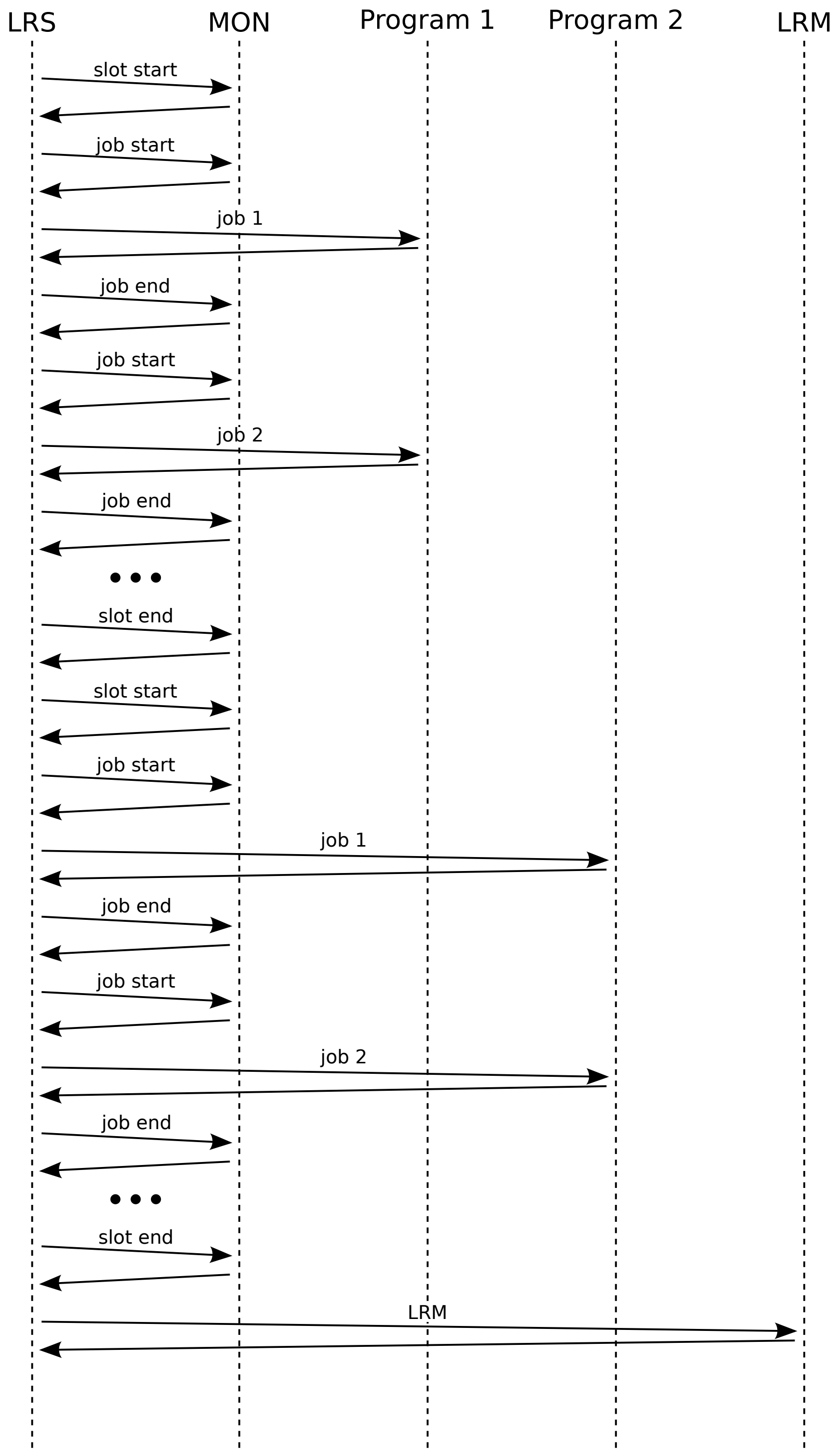

Figure 3.1: Example sequence diagram for the run-time system

3.4 Test programs

In carrying out this work, we used the following benchmark and application programs, in various combinations, to serve as the real-time and best-effort programs under test.

3.4.1 Real-time safety-critical application programs

So that our measurements would be as realistic as possible, we chose a set of real-time application programs whose run-time characteristics and behaviour would correspond to those of a typical safety-critical system.

These programs are the *flight management system* (FMS), the *display management system* (DMS), and the *sensor data provider* (SDP), each implementing functionality representative of a real safety-critical system.

3.4.2 MiBench

MiBench [14] is a set of freely available embedded benchmarks designed to represent the characteristics of commercially available programs from a number of application domains: automotive and industrial control; networking; security; consumer devices; and office automation.

Each benchmark has its own particular characteristic distribution of program instructions between integer, floating point, load, store, and branch operations. We chose the Rijndael benchmark for our tests because its encrypt and decrypt functions have a relatively large proportion of memory operations: around 50%.

We configured the Rijndael benchmark to run in a continuous loop, with a repeating sequence of 4 encrypt, 4 decrypt, and 32 verify operations. The verify operations are less memory-intensive; the intention of adding them to the loop was to produce cyclic variations in interference with a period of 5–10 major cycles of the hypervisor.

3.4.3 Memory-stressing benchmark programs

In addition, we developed a set of benchmark programs – listed below – designed to strongly exercise the memory resources shared by the processor cores, in a way that would provoke various patterns of interference and therefore variations in the running time of any concurrently-running process.

Sequential read (rs)
: 32-bit read operations on sequential 32-bit-aligned addresses in a 4 MiB[1] array in SDRAM.

Pseudorandom read (rr)
: 32-bit read operations on pseudorandomly chosen 32-bit-aligned addresses in the 4 MiB array.

[1] 1 MiB = 1024^2 bytes

Sequential write (ws)
: 32-bit write operations on sequential 32-bit-aligned addresses in the 4 MiB array.

Pseudorandom write (wr)
: 32-bit write operations on pseudorandomly chosen 32-bit-aligned addresses in the 4 MiB array.

The sequential memory operations were designed to incrementally step through memory as fast as possible (for maximum memory bandwidth), while the pseudorandom ones were designed to step across page boundaries in an unpredictable way in order to cause more cache misses. The write operations were intended to create more interference (through disruptive cache flushes) than the read operations. All were capable of having their execution time adjusted (via loop conditions) to run for some given proportion of the slot duration, or of running continuously until interrupted by the hypervisor.

3.4.4 Composite benchmark program

For later experiments, as a more memory-intensive stand-in for the real-time application programs, we used a sequence of jobs combining the memory-stressing program tasks described above, along with a *sleep* task that makes no memory access operations.

3.5 Measurements

The ultimate goal of this experimental set-up is to measure the effects of interference through shared resources (e.g. bus, shared caches, memory controllers) on a real-time application program, and the usefulness of strategies to mitigate that interference.

Since we are concerned with meeting real-time deadlines, the most important variable to measure is the real running time; we achieve this by sampling the cycle counters for each processor core.

We also want to study the memory activity, so we sample a number of related hardware counters: *data cache miss*, *data read*, and *data write*. Unfortunately, *data cache miss* appears to always return zero on our Cortex-A9 processor, and so is not useful to us.

Finally, we want to be able to estimate the progress made by each program scheduled by our run-time system, so we sample the instruction counter. On the Cortex-A9, an exact value is not available; instead we measure the number of *instructions renamed*, which gives an approximation.

All the above values are sampled via the monitoring subsystem described in Section 3.3. Measurements are taken at various levels of granularity by saving the value of each counter to shared memory at the start and end of each job, slot, and major cycle. Note that each core has its own set of hardware counters, so the measurements taken on each core are independent. Table 3.1 gives a summary of the measurement sources.

In order to exfiltrate the measurement data saved in shared memory, we implemented an optional feature in the run-time system that encodes the information

Table 3.1: Measurement sources in the Cortex-A9 Performance Monitoring Unit

Hardware counter	Description
Cycles	Number of processor cycles for which the core has been active. Given that we know the processor frequency, we can use this to calculate the real running time.
Data cache misses	Number of data cache misses. *Always returns zero on the Zynq-7000 ARM processor.*
Data reads	Number of *load* instructions architecturally executed.
Data writes	Number of *store* instructions architecturally executed.
Instructions renamed	Approximate total number of instructions executed.

stored in memory by the monitoring subsystem, and prints it to the serial console in a compact format. We are then able to extract and plot the data we are interested in with a set of scripts.

3.5.1 Interference between benchmark pairs

To begin with, we simply ran pairs of benchmark programs in parallel (one on each core) in their various combinations. This had the dual purpose of testing the experimental set up – to ensure that it was capable of adequately measuring the phenomena being studied – and giving an indication of the interference generated by different patterns of memory access.

For these tests, we ran one of the benchmark programs on core 0 of the Cortex-A9 processor, calibrated so that its running time in isolation was approximately 80% of the slot duration. We then ran the other benchmark program in parallel on core 1 ten times, stopping it after 10% of the slot duration the first time, and subsequently incrementing its running time (and therefore the duration of any interference) in steps of 10% of the slot duration, until it filled the whole slot. This procedure was repeated for each pair of benchmark programs.

3.5.2 Interference to a safety-critical application

Having established that we were able to observe and measure interference between two independent programs running on different processor cores, we proceeded to measure the effects on a set of realistic application programs (described in Section 3.4.1).

We assigned a partition to each program (FMS, DMS, SDP) and mapped these partitions to three hypervisor slots on core 0 of the Cortex-A9 processor. The slots were programmed sequentially, constituting the major cycle of the hypervisor schedule. The duration of each slot was calibrated such that the corresponding application jobs would have time to complete, even under the levels of interference being observed.

For the first run, we ran the application programs in isolation, i.e. without any other program running.

We then ran each of the four memory-stressing benchmark programs in turn on core 1. The benchmark programs were scheduled to run continuously. The running time of each of the application jobs was measured again, giving us four further sets of values.

3.6 WCET controller implementation

The next step was to reimplement the WCET controller described by Kritikakou et al. [4]. We did this by having the monitoring subsystem compare its hardware cycle counter samples with statically defined observation points specified in the run-time system configuration. This check occurs after the completion of each real-time job. If the offset of a job's end time from the start of its slot is greater than its defined observation point, a deadline overrun warning is triggered, resulting in all best-effort programs being stopped until the end of the slot.

Figure 3.2 shows an example where the the best-effort program (core 1) is stopped in two successive major cycles, due to the WCET controller issuing a deadline overrun warning at the end of the third real-time job (core 0).

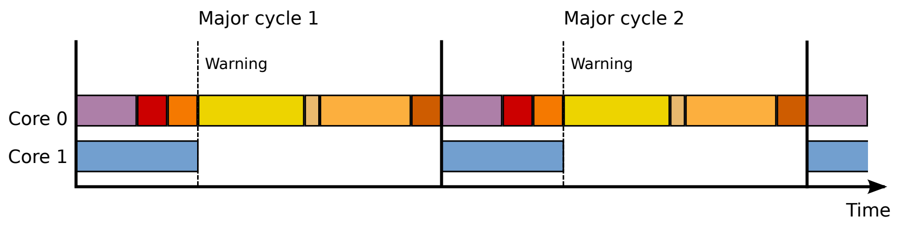

Figure 3.2: WCET controller stopping the best-effort program after a deadline overrun warning

Note that here we do not concern ourselves with how the observation points are defined in order to guarantee safe operation; questions of schedulability and calculating the remaining WCET for a real-time program slot are beyond the scope of this work. Our aim is to reproduce the WCET controller mechanism, then use this as a starting point for improving the utilisation with our QoS strategies.

3.7 Quality-of-service strategies

Having established a functional WCET controller, we were then in a position to experiment with various QoS strategies to attempt to increase the utilisation for the best-effort program. The goal here was to find a strategy that could achieve a utilisation as close as possible to that measured in isolation, and no lower than that obtained without any QoS strategy being used. Figure 3.3 illustrates how a simple QoS strategy could result in a significant improvement in utilisation by turning the WCET controller's actions to our advantage.

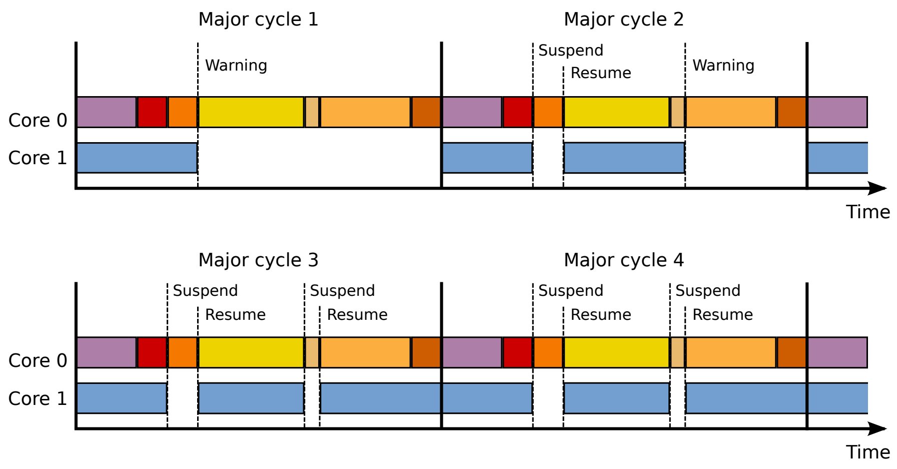

Figure 3.3: Example showing improved utilisation with a simple QoS strategy

In major cycle 1, the WCET controller stops the best-effort program after the third real-time job.

In major cycle 2, the QoS strategy programmes the job that triggered the deadline overrun warning to run in isolation; as a result, the third real-time job does not encounter interference, and an overrun warning now only occurs later in the slot, after the fifth job.

In major cycle 3, the fifth job is also programmed to run in isolation; this time no overrun warnings occur.

In major cycle 4, the pattern is repeated, the system already having reached an equilibrium, with the third and fifth jobs being run in isolation. Owing to the QoS strategy, the proportion of the time slot when the best-effort program is able to run has increased significantly compared to major cycle 1.

Of course, this is a contrived example whose outcome is a significant improvement in utilisation, but other outcomes are possible. For example, were the deadline overrun warning to be triggered by the sixth job, a simplistic QoS strategy would result in that job being programmed to run in isolation; even if the seventh and final job were then not run in isolation, it is shorter than the sixth job, and so the total running time for the best-effort program over the course of the slot would necessarily be shorter. Another limitation is that the job that triggers the deadline overrun warning may not necessarily be the job that encounters the most interference.

Table 3.2 gives a summary of the QoS strategies we evaluated. For strategy 0, no QoS action is taken. The other four strategies take various combinations of the following actions—

Action 1
If a job triggers a deadline overrun warning, programme that job to run in isolation.

Action 2
If a job triggers a deadline overrun warning, programme the job with the lowest deadline overrun metric to run in isolation. This metric is calculated by subtracting the measured duration of the job from the difference between the job's deadline and the previous job's deadline, and is

Table 3.2: QoS actions taken for each QoS strategy

	Action 1	Action 2	Action 3
Strategy 0			
Strategy 1	●		
Strategy 2		●	
Strategy 3	●		●
Strategy 4		●	●

a measure of the relative contribution of a job to a deadline overrun.

Action 3
Reset the QoS state of jobs occurring after and in the same slot as a job that triggered a deadline overrun warning.

In the event of deadline overrun warnings, the QoS strategies programme one or more real-time jobs to run in isolation, with the intention that this set of jobs reach an equilibrium over the course of a few major cycles. While the behaviour of the real-time program is by definition cyclic, the behaviour of the best-effort program may change over time. Even if its behaviour is also cyclic, the period may be much longer than a major cycle of our real-time schedule, with alternating phases of high and low interference. It is therefore necessary to reset the QoS strategy either periodically or when a change in behaviour is detected, in order that real-time jobs not be run in isolation unnecessarily.

We evaluated these QoS reset strategies—

Reset strategy 0
QoS strategy is never reset.

Reset strategy 1
Reset the QoS strategy after a fixed number of major cycles (40).

Reset strategy 2
Reset the QoS strategy if a deadline overrun warning occurs more than a fixed number of major cycles (1) after the last one, and there are currently one or more jobs programmed to run in isolation.

Reset strategy 3
Reset the QoS strategy if the number of memory stores as a proportion of all instructions drops by more than a given amount (20%) relative to the maximum recorded proportion since the last time the QoS strategy was reset.

3.8 Test procedure

For each configuration (real-time program, best-effort program, task schedule, test parameters), we launched a series of ten test runs. Each test run consists of three reference tests plus a test for each combination of the QoS strategies and QoS reset strategies.

iso	Run the real-time program on core 0 in isolation (i.e. with nothing running on core 1).
ref	Run the best-effort program on core 1 in isolation (i.e. with nothing running on core 0).
0.0	Run the real-time and best-effort programs simultaneously on cores 0 and 1, respectively, with deadline checkpoints set for each job of the real-time program, and no QoS strategies used.
Q.R	Run the real-time and best-effort programs simultaneously on cores 0 and 1, respectively, with deadline checkpoints set for each job of the real-time program, and QoS strategy Q used in combination with QoS reset strategy R.

We post-processed the serial console output for each test to extract the necessary data: per-core PMU measurements; per-slot PMU measurements; per-job PMU measurements; best-effort program progress; WCET controller actions; and QoS actions.

4
Results

4.1 Interference between benchmark pairs

We begin by observing the effects of indirect interference between pairs of the memory-stressing benchmark programs described in Section 3.4.3. Figure 4.1 shows the variation in running time of the sequential-read (rs) benchmark program running on core 0 of the Cortex-A9 processor, while under interference from each of the four benchmark programs running in turn on core 1. The interfering program on core 1 is run for 0–1000 ms in steps of 100 ms.

The increase in running time is directly proportional to the duration of the interference, up to around 800 ms. After this point, the running time of the interfering program overtakes that of the program being measured, so there is no further increase in the latter. The slowdown at 800 ms is approximately 1.3.

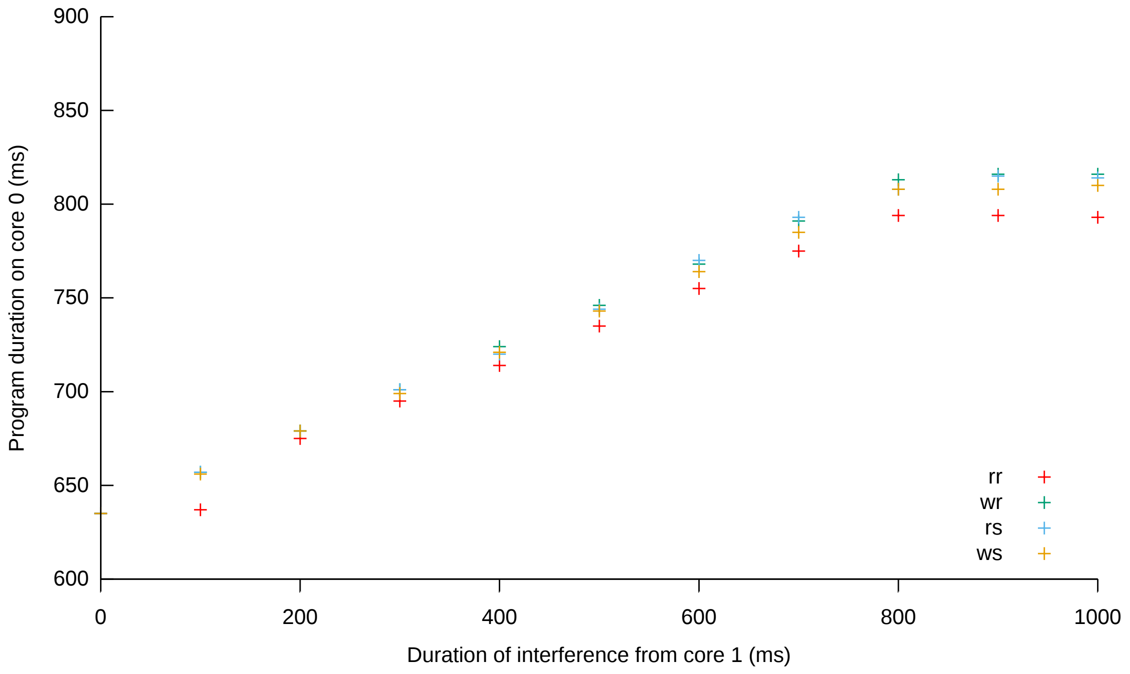

Figure 4.1: WCET of the sequential-read benchmark under interference from each of the four benchmark programs

4.2 Interference to a safety-critical application

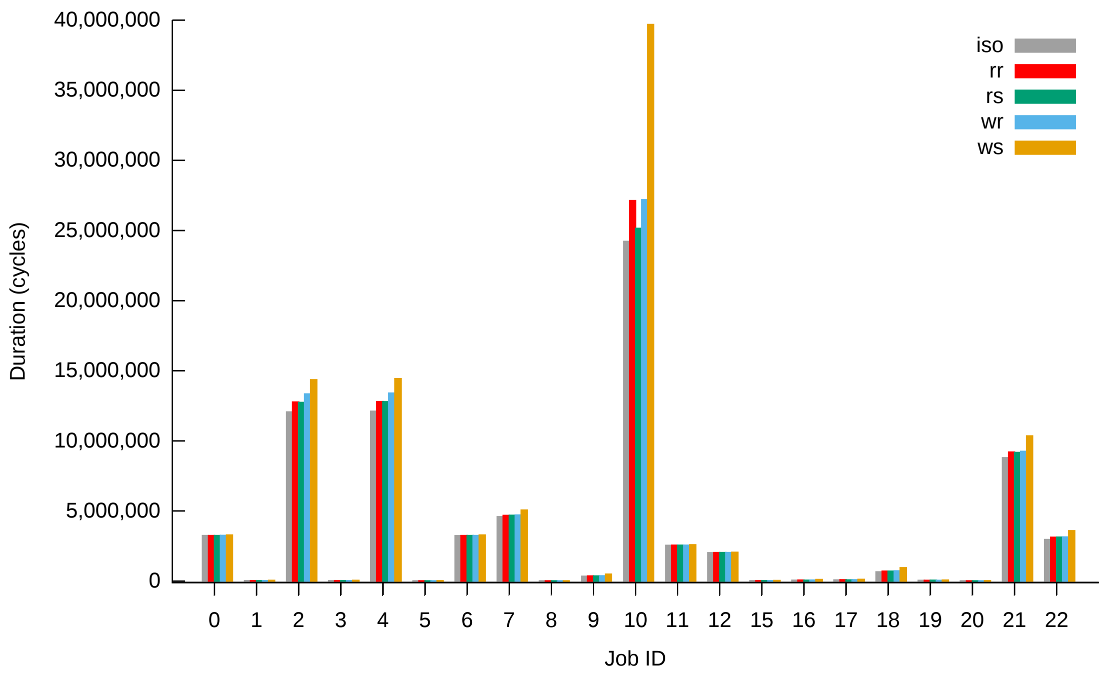

Figure 4.2: WCET of safety-critical application tasks in isolation and with interference

Figure 4.2 shows the maximum running time (in processor cycles) of the real-time application jobs within a major cycle of the hypervisor schedule. Jobs 0–4 are from the SDP, jobs 5–12 are from the DMS, and jobs 15–22 are from the FMS. In all cases, these jobs are running on core 0 of the Cortex-A9 processor.

Jobs 13 and 14 are excluded from the plot because they belong to the run-time system itself and not to the application programs.

For each job, the five running time measurements (displayed as clustered bars in the figure) correspond to five configurations—

isolation (iso)
: Only the real-time application jobs are run, and no program is scheduled on core 1.

pseudorandom read (rr)
sequential read (rs)
pseudorandom write (wr)
sequential write (ws)
: The corresponding memory-stressing benchmark program (as described in Section 3.4.3) is run continuously on core 1.

We observe that uniform interference – from a memory-stressing benchmark program running continuously on core 1 – appears not to have a uniform impact on the real-time programs (FMS, DMS, SDP). The worst interference occurs with jobs 2 and 4 (SDP), and job 10 (DMS). Neither do all of the memory-stressing programs have the same impact. The worst interference comes from the sequential memory writes (ws); memory reads have a generally lower impact.

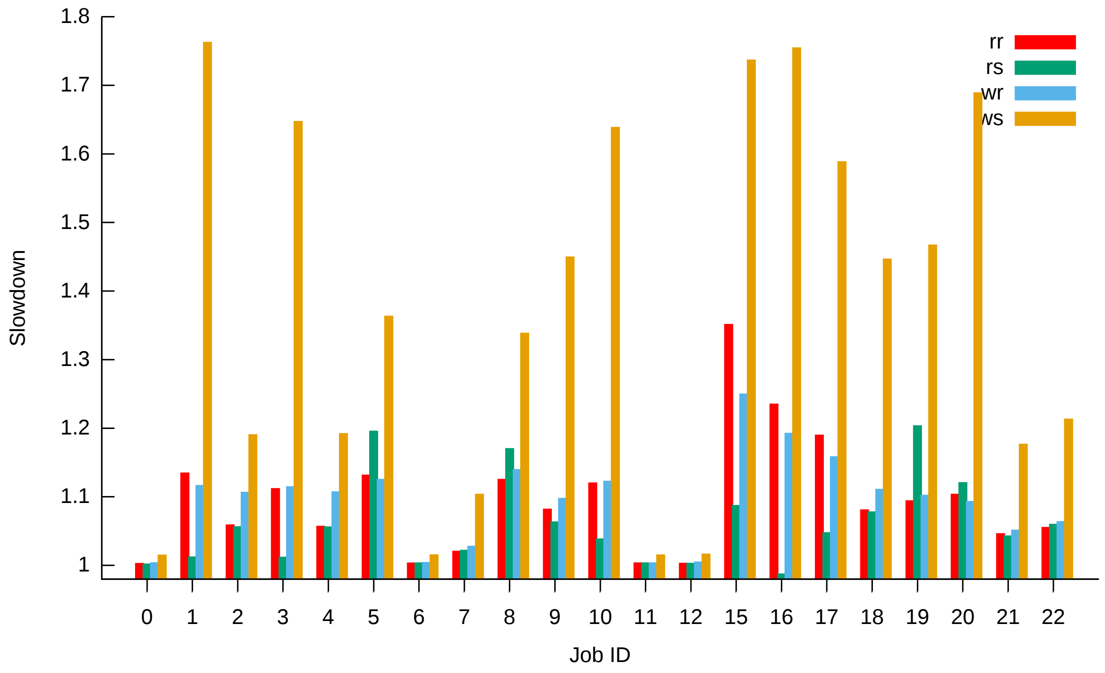

Figure 4.3: Slowdown of safety-critical application tasks with interference

Rather than looking at the absolute job duration figures, it is perhaps more instructive to consider the relative slowdown for each job. Figure 4.3 was compiled from the same data as Figure 4.2, but instead each measurement is expressed as a slowdown relative to the application program running in isolation, as given by Equation 4.1.

$$\frac{\text{WCET measured with interference}}{\text{WCET measured in isolation}} \tag{4.1}$$

Considering relative slowdown, rather than absolute job duration, gives us a better idea as to which jobs suffer the worst interference. Jobs 1, 4, and 10 (identified above) are not impacted the most. Instead, several of the shortest jobs (1, 3, 15, 16, 17, and 20) suffer the worst interference. These short jobs have limited impact in this scenario for the very reason that they are short, but this impact could be greater if there were a large number of such tasks in a real-time program.

For write-based interference, the sequential writes invariably have the greater impact, often significantly so. In terms of interference from memory reads, the impact is generally lower than for writes, but whether the sequential or the pseudorandom read operations have a greater impact depends on the real-time job being interfered with.

The deadline overrun mechanism described by Kritikakou et al. [4] ensures that excessive interference does not result in real-time deadlines being missed. Figure 4.4 illustrates our initial strategy for improving on this by cutting interference only where its impact is greatest: the memory-stressing programs running on core 1 are stopped at the start of job 10 and restarted once that job finishes.

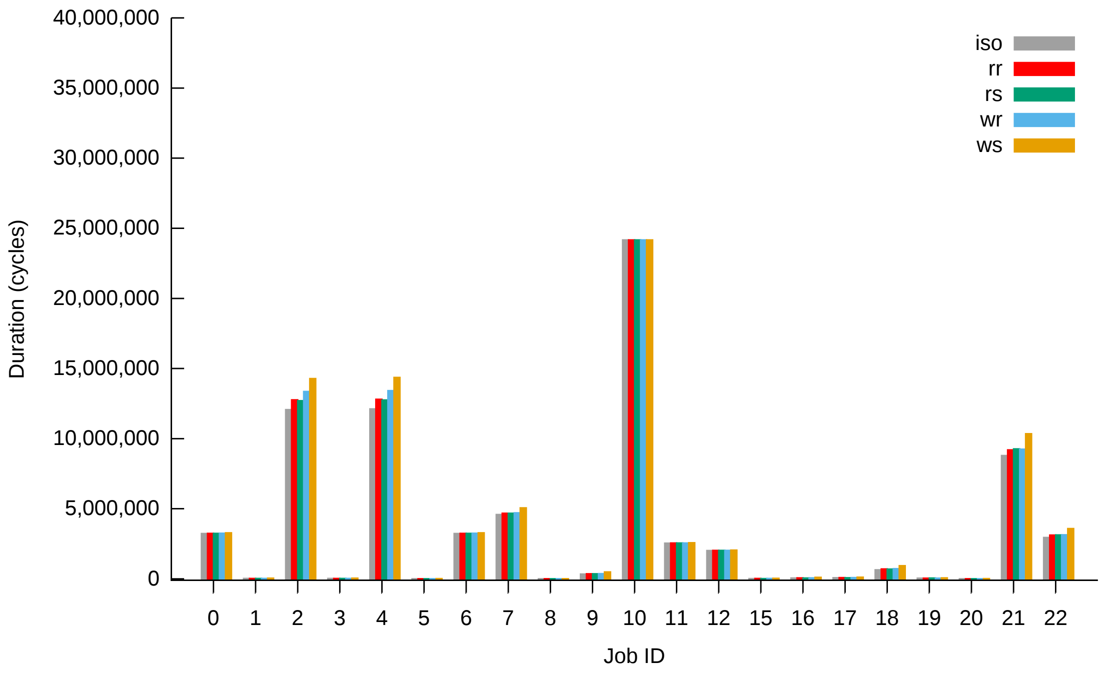

Figure 4.4: WCET of safety-critical application tasks with no interference during job 10

We made a series of 84 test runs whereby for each memory-stressing program running on core 1, we suspended it for the duration of each of the safety-critical application jobs in turn, and measured the WCET for that job. These results are aggregated in Figure 4.5: the four bars for each job correspond to the four test runs where the memory-stressing program was suspended for the duration of that job.

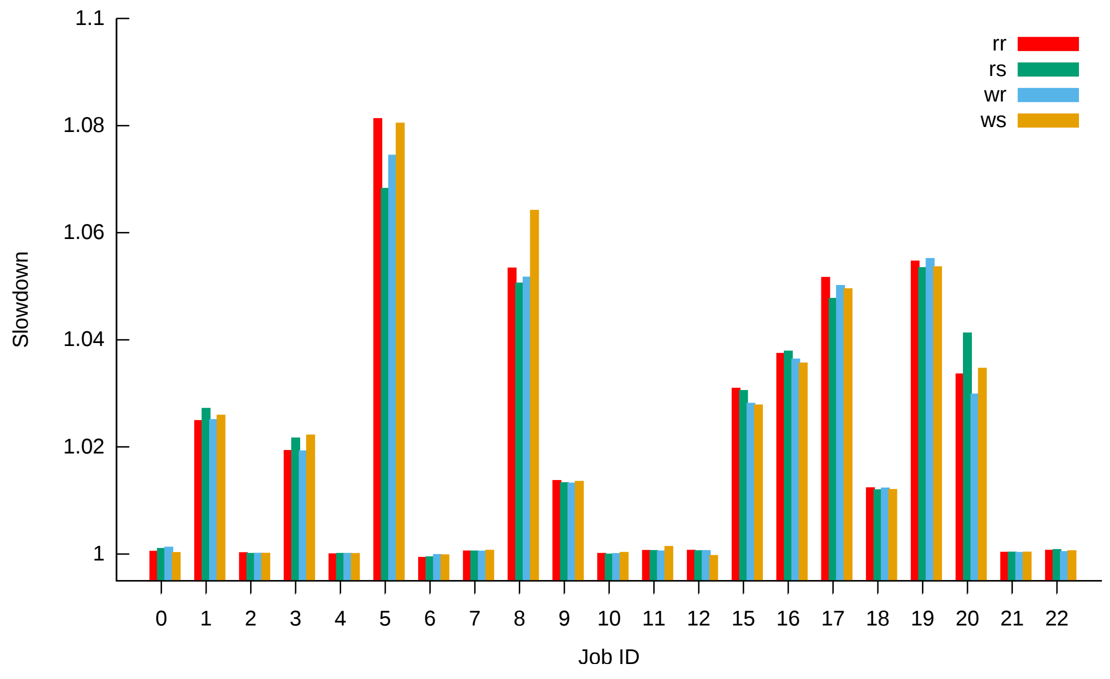

Figure 4.5: Aggregate plot of task slowdown where the memory-stressing program is suspended for the duration of the indicated job

These results show that although the technique of suspending the interfering program is effective for reducing the absolute impact on running time, the relative slowdown can still be as high as 1.08 for short tasks. One possible cause could be if the relevant hypervisor API calls[1] are not completely synchronous, resulting in some interference occurring at the start of the job. It could also be due to the cache state left by interference during the previous job. However, this observed slowdown is still much smaller than the effect of leaving the memory-stressing program running.

4.3 Comparison of QoS strategies

For each configuration, we plot the results for each of the various QoS strategies used. QoS reset strategy 1 is used in all of these configurations. The programs used for this set of results are: the composite benchmark (CB) described in Section 3.4.4 on core 0 with real-time deadlines; and the Rijndael benchmark from MiBench (see Section 3.4.2) running in a continuous loop on core 1.

The x-axis indicates the number of major cycles. (All tests were run for two hundred major cycles.)

The performance measurements are shown in the lower part of the plot, for each of the two processor cores: the left-hand y-axis indicates the number of processor cycles for which the partition is being run (blue line); the right-hand y-axis indicates the number of memory loads and stores (orange and green lines) and the total number of instructions executed (red line).

[1]`XM_suspend_partition` and `XM_resume_imm_partition`

The deadline-overrun and QoS actions are shown in the upper part of the plot, and are duplicated between cores 0 and 1 to facilitate interpretation in conjunction with the performance measurements. Vertical red lines indicate a deadline overrun warning, triggered by the job marked by a red dot. A blue dot indicates that the best-effort program was suspended for the duration of the given real-time job in a particular major cycle.

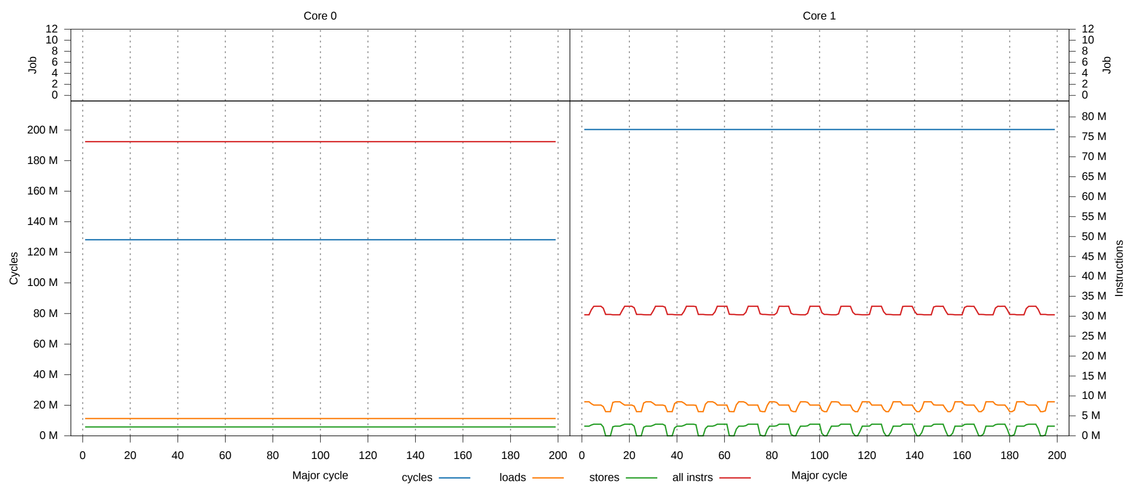

Figure 4.6: Composite benchmark in isolation, Rijndael in isolation (iso)

The first plot (Figure 4.6) is special, in that it combines two sets of results: one with the real-time program (core 0) running in isolation, and the other with the best-effort program (core 1) running in isolation. These two sets of results are combined in a single plot to aid comparison with subsequent figures.

For the real-time program, the number of instructions is constant across all major cycles. This is because it runs according to a fixed schedule, performing the same tasks in the same sequence each time, and stopping once those tasks are done. The number of cycles also remains constant, showing that there is no significant variation in running time for these tasks.

The best-effort program, however, exhibits some variation in the number of instructions per major cycle. This is consistent with the fact that it runs in a continuous loop, asynchronous with respect to the fixed schedule and to the duration of a major cycle. The number of cycles remains constant at 200 million; this corresponds to the length of the slot (0.5 s) at 400 MHz.

Figure 4.7 shows what happens when we run the two programs simultaneously, with the WCET controller turned on but no QoS strategies used. As a reminder, this configuration is the baseline for our work, upon which we must obtain an improvement.

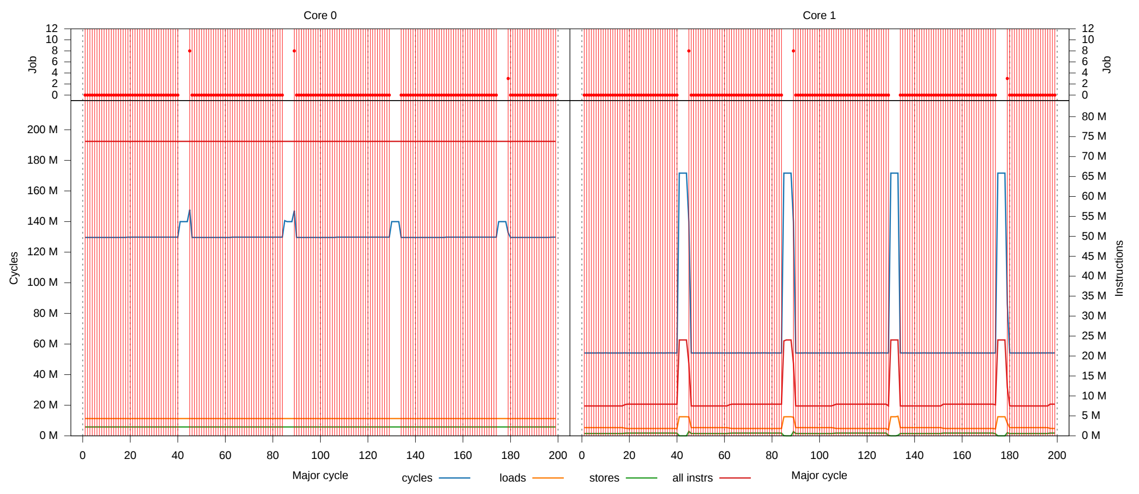

Figure 4.7: Composite benchmark (CB) + Rijndael, no QoS (QoS 0.0)

Immediately, we see from the vertical red bars that a deadline overrun warning was triggered by the WCET controller during most major cycles. Most of these occur after the first job, although during three major cycles the warning is triggered after job 4 or job 8 instead.

Although the WCET controller ensures safe operation by stopping the best-effort program when necessary, we see that this has a significant impact on the latter's performance: for most major cycles, its instruction count (the red line on the lower right part of the plot) drops to less than a third of the value measured when running in isolation. This is unsurprising given that it is being stopped by the WCET controller after only one real-time program job, and is unable to resume until the start of the following major cycle.

For the major cycles where there is no deadline overrun warning, the instruction count for the best-effort program is significantly higher at around 60 million, albeit still lower than the values obtained in isolation. The cycle count for the real-time program is higher than in isolation; this is consistent with cache interference from the best-effort program slowing down the real-time jobs.

Figure 4.8 shows the result of applying QoS strategy 1, whereby each real-time job for which a deadline overrun warning occurs is programmed to run in isolation in subsequent major cycles. QoS reset strategy 1 is also applied here, meaning that the QoS policy is reset every forty major cycles.

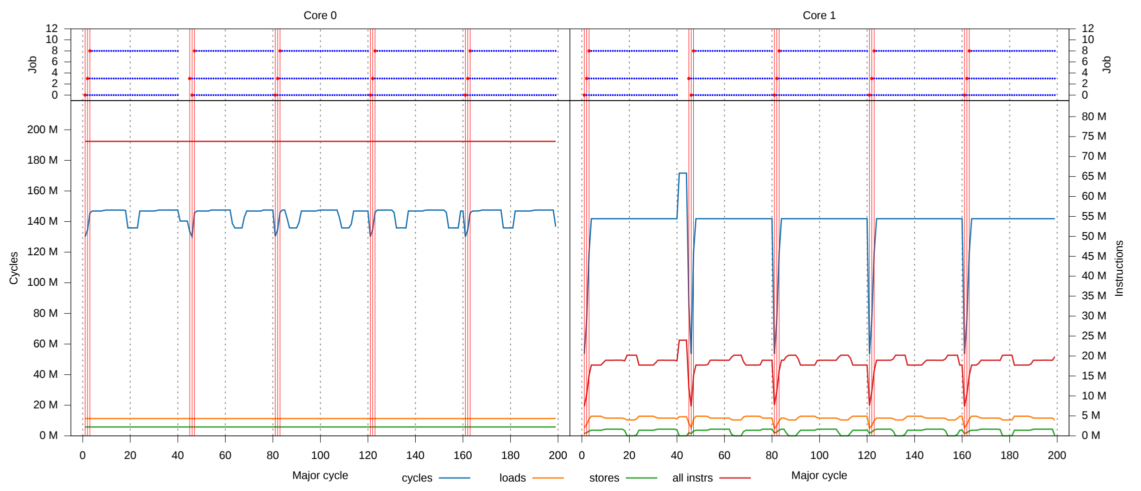

Figure 4.8: CB + Rijndael, QoS strategy 1, QoS reset strategy 1 (QoS 1.1)

As before, we see that in major cycle 1 there is a deadline overrun warning after the first job. As a result, the best-effort program is stopped by the WCET controller for the remainder of the major cycle, and the instruction count remains at around 20 million.

At the end of major cycle 1 the QoS policy is updated, and instructs the scheduler to suspend the best-effort program for the duration of the job that triggered the deadline overrun warning (job 0 of the real-time program) in subsequent major cycles. In major cycle 2, the deadline overrun is therefore avoided at this point. When the scheduler resumes the best-effort program after job 0, it is able to run during jobs 1, 2, and 3, before being stopped by the WCET controller due to a deadline overrun warning after job 3.

In major cycle 3, the best-effort program is suspended for the duration of jobs 0 and 3 (both of which triggered deadline overrun warnings), and is able to run during jobs 1, 2, and 4 to 8. It is again stopped by the WCET controller after job 8.

In major cycle 4, the QoS policy reaches an equilibrium: the best-effort program is suspended during jobs 0, 3, and 8, but is able to run for the remaining duration of the time slot without a deadline overrun warning occurring. This equilibrium continues until major cycle 40, when the QoS reset policy causes the QoS policy to be returned to its initial state (i.e. with no real-time jobs programmed to run in isolation).

The pattern of deadline overrun warnings, separated by longer periods of QoS policy stability, repeats itself roughly every forty major cycles, in line with the QoS reset policy. The pattern is not exact, though: immediately after major cycle 40, both programs are able to run concurrently for several major cycles without a deadline overrun warning being triggered. This corresponds to a temporary lull in memory store operations by the best-effort program, in turn resulting in a lower level of interference to the real-time program. And as the number of memory stores ramps up again, the first deadline overrun warning occurs after job 3, rather than after job 0.

Apart from the major cycles when deadline overrun warnings occur, or shortly after major cycle 40, the instruction count for the best-effort program is around 40–

50 million, approximately double the value without any QoS strategy being used.

Figure 4.9 shows the result of applying QoS strategy 2, whereby for each deadline overrun warning, the real-time job that suffered the most interference is programmed to run in isolation in subsequent major cycles. QoS reset strategy 1 is also applied.

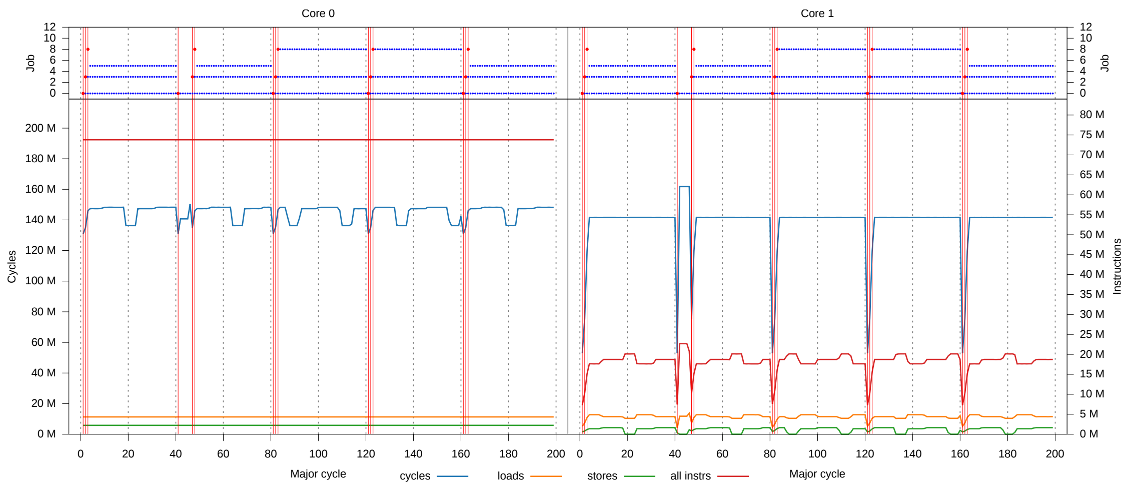

Figure 4.9: CB + Rijndael, QoS strategy 2, QoS reset strategy 1 (QoS 2.1)

We see that after the deadline overrun warning in major cycle 3, job 5 is programmed to run in isolation, rather than job 8. This indicates that job 5 encountered more interference than job 8, despite the latter being the immediate source of the deadline overrun warning. Between major cycles 80 and 160, job 8 is the one selected to run in isolation; this suggests that the time-varying behaviour of the best-effort program affects the relative level of interference experienced by jobs 5 and 8.

Despite the less simplistic nature of QoS strategy 2, there is no readily discernible difference in the instruction count for the best-effort program when compared with QoS strategy 1.

Figure 4.10 shows the result of applying QoS strategy 3. This is identical to strategy 1, except that when a deadline overrun warning is triggered, the QoS policy is reset for jobs scheduled in the same slot after the triggering job.

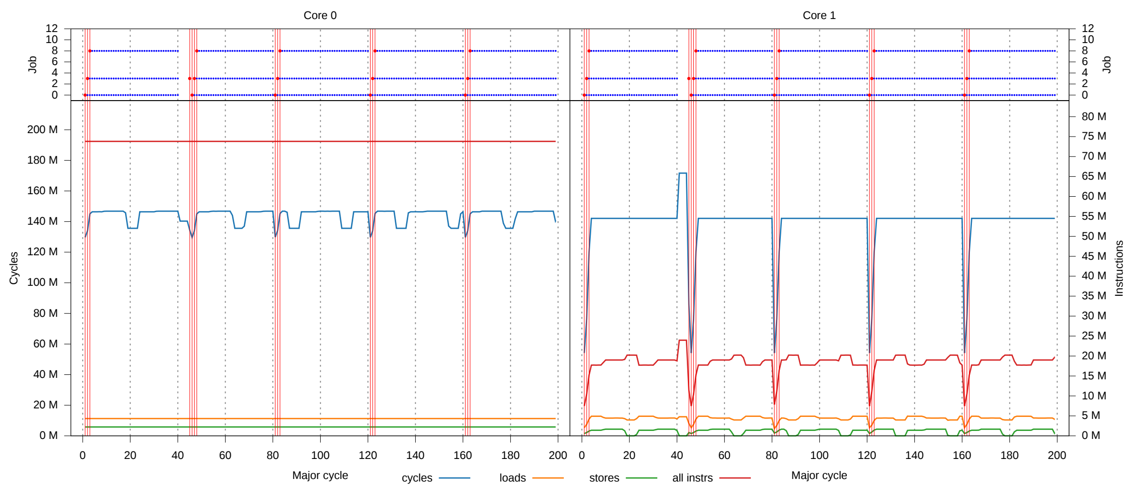

Figure 4.10: CB + Rijndael, QoS strategy 3, QoS reset strategy 1 (QoS 3.1)

In practice, this strategy gives almost identical results to QoS strategy 1. The only readily discernible difference concerns the cluster of deadline overrun warnings after major cycle 40: the first overrun of the cluster is triggered by job 3, causing that job to be programmed to run in isolation, but the second overrun occurs earlier, after job 0. QoS strategy 3 resets the QoS settings for later jobs, so in this case job 3 is no longer programmed to run in isolation, only job 0.

However, the next major cycle sees a deadline overrun triggered again by job 3, so the only real difference is that this strategy takes an extra major cycle to reach equilibrium.

Figure 4.11 shows the result of applying QoS strategy 4. This is identical to strategy 2, except that when a deadline overrun warning is triggered, the QoS policy is reset for jobs scheduled in the same slot after the triggering job.

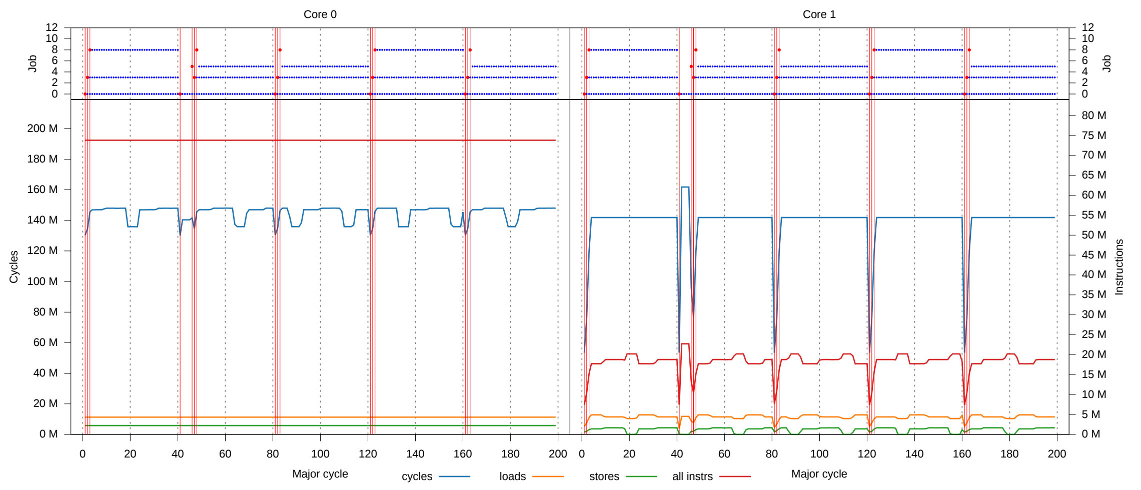

Figure 4.11: CB + Rijndael, QoS strategy 4, QoS reset strategy 1 (QoS 4.1)

In this case, the outcome shows the similarity to QoS strategies 2 and 3: the algorithm sometimes chooses job 5 to run in isolation and at other times chooses job 8; the cluster of overruns after major cycle 40 also takes an extra major cycle to reach equilibrium.

4.3.1 Comparison of QoS reset strategies

For each configuration, we plot the results for each of the remaining QoS reset strategies used. QoS strategy 4 is used in all of these configurations.

Figure 4.12 shows the result of applying QoS reset strategy 2, whereby the QoS strategy is reset if a deadline overrun warning occurs after a gap of at least one major cycle without one, and there are jobs currently programmed to run in isolation.

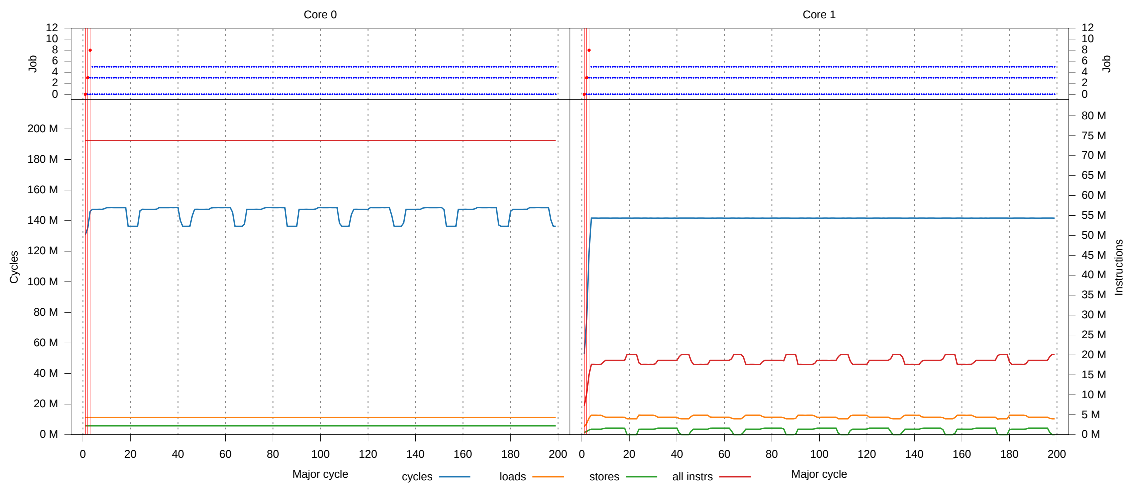

Figure 4.12: CB + Rijndael, QoS strategy 4, QoS reset strategy 2 (QoS 4.2)

Once an equilibrium is reached in the fourth major cycle, no further deadline overrun warnings occur. This demonstrates the limited utility of this reset strategy, in that it only takes into account an increase in interference to jobs not programmed to run in isolation, and does not react when the level of interference drops.

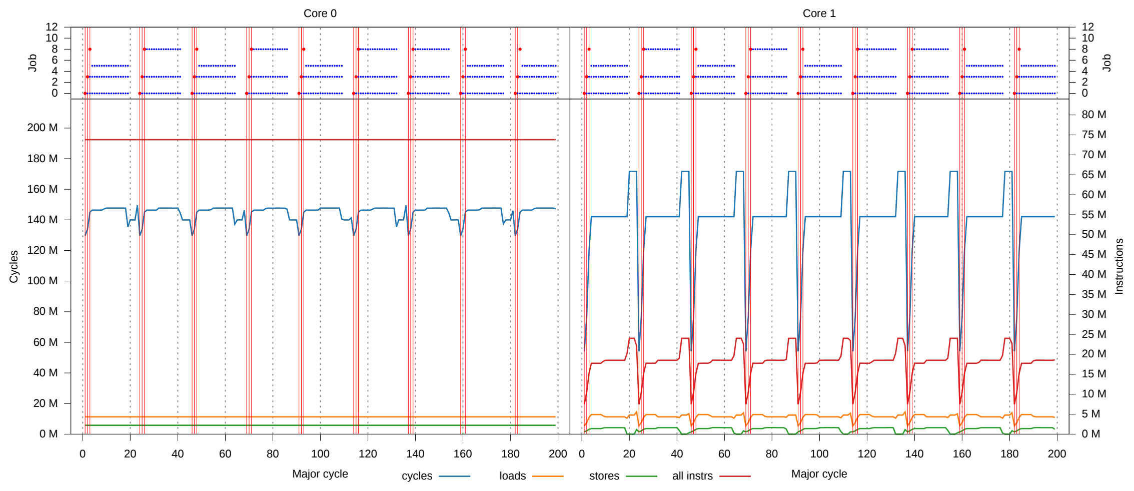

Figure 4.13: CB + Rijndael, QoS strategy 4, QoS reset strategy 3 (QoS 4.3)

Figure 4.13 shows the result of applying QoS reset strategy 3, whereby the QoS strategy is reset if the number of memory stores as a proportion of all instructions drops by at least 20% from its maximum value since the QoS strategy was last reset.

We see that this reset strategy is effective at detecting changes in the behaviour of the best-effort program resulting in a drop in interference. This allows the best-effort program to progress unimpeded during phases of low interference, while ensuring that the QoS strategy continues to be effective during phases of higher interference.

4.3.2 Evaluating the impact on utilisation

So far we have made a qualitative comparison of the various QoS strategies and QoS reset strategies. However, since our aim is to improve utilisation, we must consider the quantitative impact that each of these strategies has. Figures 4.14 and 4.15 show the relative change in performance (cycle and instruction counts) for each QoS setting, averaged[2] over ten test runs, for the real time program and the best-effort program, respectively. The y-axes indicate relative change from the measurements obtained while running in isolation. QoS setting $Q.R$ represents QoS strategy Q being used in combination with QoS reset strategy R.

The real-time program operates to a fixed cyclic schedule, so the instruction count is identical in all cases. The running time (cycle count) increases by a factor of around 1.02 with no QoS strategies used (QoS setting 0.0); this is consistent with interference from the best-effort program delaying the memory operations of the real-time program. The running time increases by a factor of around 1.12–1.13 with the other QoS settings; this is consistent with the best-effort program being active for a larger proportion of each slot, thereby causing more interference (and more delay) to the real-time program.

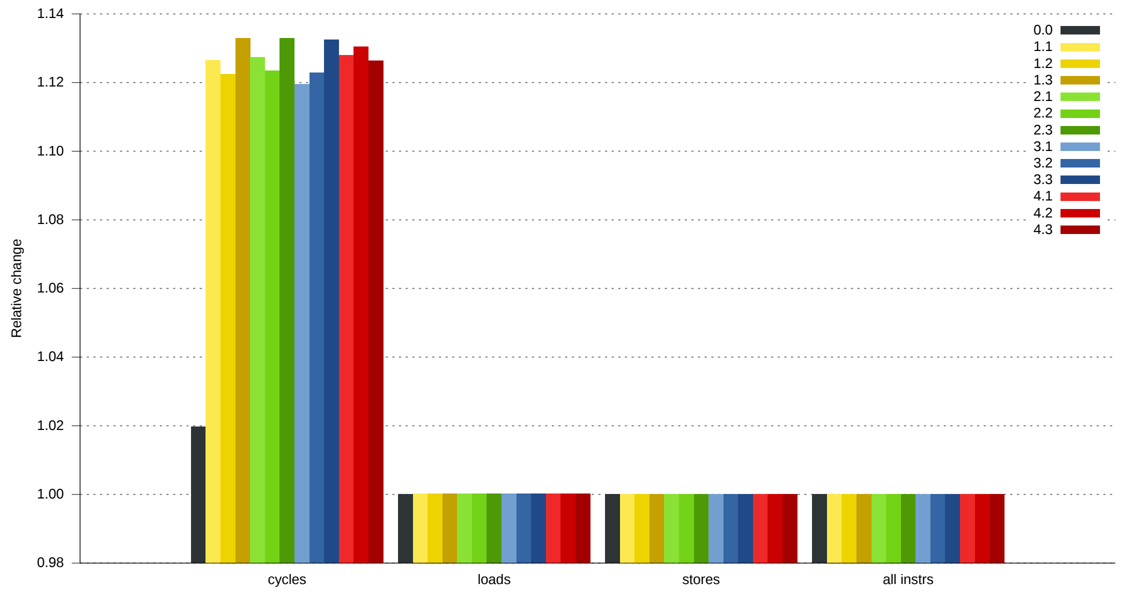

Figure 4.14: Relative change in average performance with each QoS setting (core 0)

[2]Since we are comparing ratios, this is a geometric average.

For the best-effort program, we are trying to get as close as possible to the performance obtained in isolation, and in any case to improve on the performance obtained without employing any QoS strategies. We see that without QoS, the cycle count (and thus processor utilisation) drops to around a third of its value measured in isolation. But with the other QoS settings, we see a cycle count of around two thirds of that measured in isolation, a considerable improvement. Furthermore, despite the observed behavioural differences between the various QoS strategies, we note that the quantitative improvement in utilisation due to our QoS policies is very similar, whichever strategy is used.

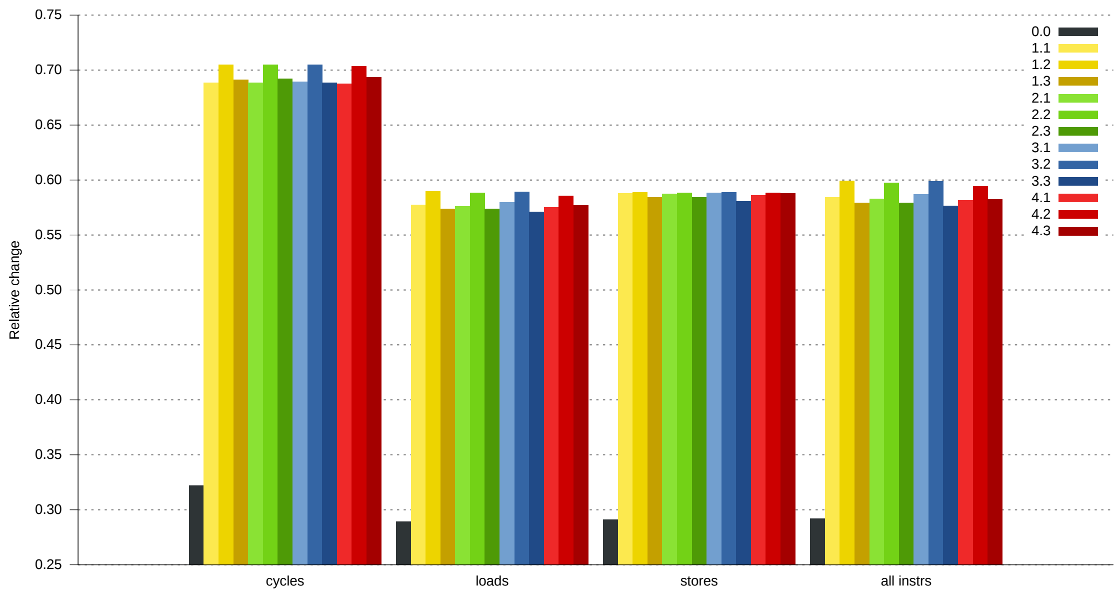

Figure 4.15: Relative change in average performance with each QoS setting (core 1)

5 Discussion

The first set of results (interference between benchmark pairs) confirms that a memory-intensive program's running time increases when subject to indirect interference through the shared memory resources of a dual-core ARMv7 processor.

The second set of results (interference to a safety-critical application) confirms the significant impact on the running time of a realistic real-time application program, and that the observed slowdown is dependent on the characteristics of both the real-time jobs and the interfering program. Our experiments indicate a worst-case slowdown of 1.7–1.8 for a typical safety-critical application. By suspending the interfering program for the duration of a given real-time job, we manage to reduce the worst-case slowdown to less than 1.1, demonstrating the basis for the WCET controller and our proposed QoS mechanism.

Moving on to the evaluation and comparison of QoS strategies, we were first able to implement the WCET controller, and show that although it is able to keep the real-time program's running time within statically defined limits – the real-time deadlines – the available running time for the best-effort program running in parallel drops significantly. Indeed, this technique may have a drastic effect on processor utilisation if the best-effort processes are regularly stopped near the beginning of a time slot as a result of high levels of interference.

Our QoS technique is designed to avoid – or at least reduce – this under-utilisation by stopping the best-effort program for the duration of a real-time job that incurs high levels of interference, and then restarting it once that real-time job has finished. We show that such a technique could indeed increase utilisation for the best-effort program, our experiments indicating an improvement in available running time by a factor of two over the WCET controller alone.

On the basis of our experiments, there is very little to differentiate the various QoS strategies and reset strategies, all of them resulting in very similar performance (best-effort program running time) relative to that observed in isolation. This would appear to suggest that the details of the QoS strategy are less important than the basic idea behind them all, which is to suspend the best-effort program for enough periods of high interference that no real-time deadline is in danger of being overrun.

However, we would caution against extrapolating this data set by assuming that our proposed QoS strategies are effective in all – or even most – cases. For example, the experimental results show significant differences between the QoS reset strategies in terms of how reactive they are to variations in best-effort program behaviour. QoS reset strategy 3 is the only one to detect and act upon a drop in interference, even if the measured improvement in utilisation is slightly worse than for QoS reset strategy 2, which never triggers a reset in our experiments. One could easily imagine a scenario that starts with a phase of high interference, and then settles into a long phase of low interference; QoS reset strategy 2 would not act on this, however long the low-interference phase, whereas QoS reset strategy 3 would act.

Another factor suggested by the results is that the potential benefits of resetting the QoS strategy, either regularly or after a change in the pattern of interference, might be tempered by the loss in utilisation due to the run-time system taking several major cycles for the QoS state to stabilise afterwards. This effect could be mitigated by having shorter major cycles, but the length of the major cycle is itself dependent upon the real-time program schedule, and thus not necessarily possible to manipulate.

In conclusion, we have demonstrated the potential value of using QoS strategies to obtain significant improvements in processor utilisation. However, more experiments are needed with a large number of different scenarios to explore the impact of the various QoS strategies and reset strategies under a wide range of run-time conditions. Our experience shows that this will be very time-consuming, but necessary to get a fuller picture of when and to what degree applying these strategies is advantageous.

5.1 Future work

In addition to carrying out more experiments to evaluate the QoS strategies and reset strategies in a wider range of scenarios, there is another dimension in which we would like to see it extended, namely the number of processor cores.

Our work so far only considers a dual-core processor, but the problem of interference through a shared memory hierarchy is equally applicable to similar processors with more than two cores. Consider the case where a real-time program runs on one core of an eight-core processor, with one best-effort program running on each of the other cores: the real-time program can encounter interference from each of the best-effort programs, potentially resulting in a deadline miss. The use of a WCET controller may result in far worse utilisation than with a dual-core processor, because all seven best-effort programs would be stopped for the remainder of the major cycle in the event of a deadline overrun warning. With our QoS strategies, this outcome could be improved, but do we suspend *all* of the best-effort programs for the duration of the real-time jobs encountering the most interference? Only some of them? Which ones? If only some of the best-effort programs are stopped, should different combinations of best-effort programs be stopped in successive major cycles so as to spread the impact on utilisation?

It was our intention to test some of these ideas on the NXP QorIQ T4240, which has twelve processor cores. However, a port of the hypervisor software, Xtratum, was not available for the QorIQ architecture until the end of this project, so there was insufficient time to port our run-time system and test infrastructure.

Bibliography

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[2] J. Bin, *Controlling execution time variability using COTS for safety-critical systems*. PhD thesis, Université Paris-Sud, Orsay, France, 2014.

[3] J. Bin, S. Girbal, D. Gracia Pérez, and A. Merigot, "Using monitors to predict co-running safety-critical hard real-time benchmark behavior," *International Conference of Information and Communication Technology for Embedded Systems*, 2014.

[4] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. Gracia Pérez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, (New York, NY, USA), pp. 139–148, ACM, 2014.

[5] A. Burns and R. I. Davis, "Mixed criticality systems – a review," tech. rep., Department of Computer Science, University of York, York, UK, January 2016.

[6] B. Hu, K. Huang, G. Chen, L. Cheng, and A. Knoll, "Adaptive runtime shaping for mixed-criticality systems," in *Proceedings of the 12th International Conference on Embedded Software*, EMSOFT '15, (Piscataway, NJ, USA), pp. 11–20, IEEE Press, 2015.

[7] R. Schneider, D. Goswami, A. Masrur, M. Becker, and S. Chakraborty, "Multilayered scheduling of mixed-criticality cyber-physical systems," *Journal of Systems Architecture*, vol. 59, no. 10, Part D, pp. 1215–1230, 2013.

[8] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (Philadelphia, PA, USA), pp. 55–64, IEEE, 2013.

[9] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, (Dresden, Germany), pp. 263–272, 2009.

[10] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J. J. Metge, "Xtratum: an open source hypervisor for TSP embedded systems in aerospace," *Data Systems in Aerospace (DASIA)*, 2009.

[11] Xilinx, Inc., *Zynq-7000 All Programmable SoC Technical Reference Manual*, February 2015. UG585 (v1.10).

[12] ARM, Ltd., *Cortex-A9 Technical Reference Manual*, 2012. Revision: r4p1.

[13] ARM, Ltd., *ARM Architecture Reference Manual, ARMv7 and ARMv7-R edition*, 2014. Issue: C.c.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: a free, commercially representative embedded benchmark suite,” in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC ’01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

A
Statement of contribution

The experimental work presented in this dissertation was carried out in the context of a research team at Thales, in Palaiseau.

A.1 Implementation

The run-time system was developed principally by Daniel Gracia Pérez and by me at Thales, with contributions from our research partners at ONERA, *Technische Universität Kaiserslautern*, and *Universität Siegen*. My contributions were focused on the aspects most relevant to this work: worst-case-execution-time controller; monitoring subsystem; quality-of-service strategies and reset strategies; memory-stressing benchmark programs; composite benchmark program; and test automation.

A.2 Dissertation

The text, figures, and tables in this document are my own work, and were produced for the purposes of this dissertation.