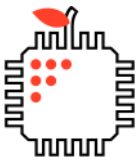
Multi-threaded processor for space applications

Final Report on ESA contract 4000106331

Chris Jeshope, Jian Fu and Qiang Yang



Techne
Consulting

Table of Contents

1 Executive Summary

This report describes work undertaken under ESA contract 4000106331 by Techne Consulting Ltd. in collaboration with the University of Amsterdam. The foundation for this work is the multi-threaded, multi-core platform emulation called the Microgrid, which was developed by the University of Amsterdam and partners in the EU Apple-CORE project (http://www.apple-core.info).

Work under this contract included the modification of this emulation platform in order to investigate two aspects of computing systems of interest to staff at ESA, namely, the use of priority threads to support real-time (RT) tasks and the use of thread-level redundancy to enhance system reliability. The report is divided into two main sections that cover these two aspects of the work.

Simulation of RT threads was performed using two synthetic benchmarks, one small and computationally intensive and the other larger and memory intensive. These are an attempt to model control and communications algorithms respectively. The benchmarks were executed while a background task (a large FFT) was running. For both benchmarks the following general results were observed:

1. Running the RT benchmark with priority threads always improved both execution speed and jitter, typically by between a factor of between 2 and 4 times for both in all experiments.
2. When the background task was run on 4 cores it was clear that there was interference from the background task through use of non-prioritised shared resources (e.g. floating-point unit and particularly the L2 cache and memory interface), however despite this, using priority threads for the RT task always improved the RT task performance and jitter.
3. When both RT and background task were restricted to a single core, the execution time of the RT task was only marginally slower than its execution stand-alone.

In conclusion, using dual thread priority in the Microgrid core gave significant improvements in both run time and jitter, however when background tasks placed stress on the memory system the benefits were not so obvious. Even so there was always at least a halving of the RT task execution time (both maximum and average over multiple executions) and jitter, when using priority threads over non-priority threads. In addition there was no significant impact on the execution time of the background task. It is recommended that further investigation be undertaken on a prototype single core with real-time threads.

For reliability, this report outlines and implements, in the Microgrid, a scheme for automatic, user-defined thread replication and execution. As expected, it is shown that there is an impact for executing double the number of instructions through duplication but this overhead is always less than 100% and sometimes significantly so, due to the multithreaded nature of the core. It is shown that the less efficient the algorithm, the lower the overhead for duplicating the threads.

Although not implemented in the emulation platform, the report also outlines a strategy for fault recovery. This uses a combination of hardware and software support, the latter provided by the programmer of the application. The hardware support does not require any additional hardware but relies on the master/redundant thread comparison buffer used in this scheme. It delays writes to L2 cache from simple threads with few writes and is able to restart the thread if an error is detected. Because of the nature of the programming model for the Microgrid, where all independent computations are exposed as threads (e.g. all independent loop iterations), this covers the common case. For threads that are not "simple", when an unrecoverable error is detected the remaining work in the region being protected is aborted a return code is passed to the code through the synchronisation primitive, this allows the programmer to write code in a way that allows efficient check pointing on these regions of code.

This work will be continued by Fu Jian at the University of Amsterdam and further results on the implementation of a complete fault detection/recovery and simulation using fault injection may be of future interest to ESA.

2 Implementation of Priority threads in MGSim

2.1 Background and motivation

2.1.1 The Microgrid

This Section of the report describes the implementation details of the addition of a static thread-level priority scheme for the *Microgrid*. The Microgrid is a configurable and scalable platform comprising Dynamically-scheduled RISC (*DRISC*) cores. The cores support hardware *Micro-threads,* which are scheduled using dataflow synchronisers, the core's registers. Currently the Microgrid exists as a full system simulation with a cycle-accurate timing simulation for most operations (I/O and some on-chip network operations are idealised). Details of the simulator platform *MGSim* can be found in [1].

One of the key features of the Microgrid is its ability to manage as well as schedule its threads in hardware, using additions to the core's ISA. This is the equivalent of implementing parts of the operating system's kernel in hardware and hence this work targets both architecture and operating system issues for many-core systems. A major characteristic of the Microgrid approach is the automatic and fair distribution of software parallelism to hardware resources (mapping software to hardware threads), giving a resource agnostic programming model. This together with the efficient thread scheduling using data flow synchronisers and hardware supported selective barrier synchronisation results in a very efficient architecture. If many hardware threads are implemented on each core it is possible to hide long latencies in instruction execution and yield high-core utilisation, even when code is distributed across many cores. It is not uncommon for regular applications to an achieve average IPC of 0.9 for the single-issue core across even large numbers of cores.

2.1.2 Concurrency and concurrency management in the Microgrid

In the Microgrid, any task can be abstracted as families of threads, structured as a hierarchical tree denoting different granularities of concurrency. This composition is flexible and is made according to explicitly expressed concurrency creation in the dynamic execution of a program. During execution, a part of the concurrency tree, perhaps all, is mapped semi-automatically[1] to a cluster of one or more cores, called a *Place* in Microgrid terminology. Places must be requested and allocated to the thread at the root of the distribution. Software threads from families at different levels in the tree are then mapped to hardware threads to do computation while being scheduled with the goal of filling all of those pipelines as far as possible. The entire task is concluded with a barrier synchronisation on all threads at the top level, which in turn may be dependent on other lower level barrier synchronisations.

---

[1] We say semi-automatically as the program must specify limits to the number of hardware threads per core in each level of the tree in order not to exhaust resources too soon.

For a specific family, the procedure is generalized as contracting the minimum number resources to assure execution (*family allocation*), followed by obtaining the desired number hardware threads with contexts (*family creation*) and finally by normal thread execution, switching into or out of the pipeline on data dependencies. In family creation, if the number of hardware threads required is less than the number of software threads, either by program-specified limits or by resource constraints, the hardware iterates the software threads sequentially through the available resources.

There are two main issues here that are relevant to prioritising threads. Family allocation is non-pre-emptive and if no resources are available then the alternatives are to fail or wait. Note that with cyclic dependencies, the latter can induce deadlock. The other is that scheduling to the pipeline is fair. Both are inadequate for the real-time requirements and are key factors in the design of the priority scheme implemented.

2.1.3 Real-time and multi-core systems

The rudimentary role of real-time systems is in responding to the most urgent task and completing its execution either within the constrained period (hard real time) or with mild tardiness (soft real time) and where the degrees of imminence of a task to its deadline are often termed priorities. To meet a deadline, the core may often devote all its resources to the task with the highest priority until it is complete in order to get rid of any interference from other tasks. It may then resume the interrupted job or jobs.

As real-time systems start using multi-core processors to handle applications with inherent concurrency for better performance or even for power reduction, this all-for-one strategy used in single-core processors is less efficient in both resource utilization and system throughput. On the one hand there is no guarantee of full-time occupation of the privileged task over multiple cores and on the other hand some other job may be seriously delayed or violate its timing requirement if this strategy is followed. Alternative strategies may share the resources, usually at a core level (thread level on thread-per-core software models). For example a priority level may be given a core exclusively until completion while still allowing other priority tasks to use other cores. The concerns now change as typically cores share resources such as elements of the memory hierarchy, busses, networks etc., which may cause interference with the priority task. It is therefore a difficult trade-off in multi-core systems between real-time assurance and other metrics, which challenge task management policy.

Up to now, studies for real-time, multi-core platforms usually fall into two categories:

1. Task scheduling policies mostly at the software level, i.e. implemented in the operating systems. E.g. [2] and [4] schedule to accelerate real-time tasks via full use of shared caches . [3] and [5] explore real-time scheduling on heterogeneous systems and [6] focuses on energy efficiency. However, most have one thing in common, i.e. that cores are task exclusive (the above mentioned share at core level) and a task rescheduling implies a migration between cores entailing higher costs than thread switches on the same core.

2. Dedicated architecture design ensures even tighter bounds for hard real-time jobs. It enables the co-execution of all kinds of tasks on the same core but equips dedicated or prioritized hardware resources for real-time tasks. [7] to [9] are examples built on simultaneous-multithreading (SMT) cores which seem to be popular in this research area to exploit TLP as well as ILP.

However priority tasks are managed, whether in software or in hardware, there are two approaches to scheduling priority tasks in a system. The first is the static naming of priority in tasks. For example the system designer associates priority levels to certain tasks and these priority levels determine the scheduling of the task at run time. This is the cornerstone that provides fundamental support for real-time tasks, i.e. preferred resource and concurrency management. On top of this, it is possible to add dynamic priority where deadline monitoring can be used to tune a task's priority level during execution. Although the latter has great potential in a fine-grain multi-threaded schedule, this report focuses on static prioritisation.

2.1.4 Research issues

This report investigates the use of priority in the Microgrid to study embedded, real-time systems in the context of multi-core, multi-threaded processors. We restrict the results of this study to the use of static priority. This report outlines the changes made to MGSim to support priority in concurrency management and scheduling in the Microgrid. These changes are used to investigate application strategies applicability to this domain through the appropriate use of priority in combination with space sharing.

As already mentioned, the Microgrid does not support pre-emption (at least as it is generally understood). Thus, even on a single core, it is not possible to use the strategy described above where a priority task acquires all of the processor's resources until it complete. The real-time task would have to wait an unbounded time to acquire a resource. This approach is also not desirable. The virtualisation of the core through the fine-grain interleaving of multiple hardware threads means that unless the real time task were able to support sufficient threads to tolerate latency, the core would become very inefficient. This may be a tolerable solution to achieve a deadline but it not is energy efficient and hence not an optimal solution. It is preferable to prioritise one or more virtual cores (threads) over the others in such a way that resources are guaranteed at family allocation and prioritised at family creation and scheduling. That allows time-sharing priority tasks to be achieved without pre-empting an existing tasks. This is therefore the approach taken in this study.

The major research question for this work is therefore whether it is possible to achieve a similar result by prioritising a real-time task and continuing to timeshare it, compared to what would be when making the core task exclusive.

The natural resource model for the Microgrid, i.e. considering more than one core, is one of space sharing. Hence a secondary research question is what impact does sharing resources such as cache, FPU etc. have on real-time tasks allocated a complete core's resources.

2.2 Implementing Priority in the Microgrid

As a foundation, the priority levels held in the Microgrid is a system-wide variable and is configurable in profiles, which conforms to the current convention in the execution of MGSim. The priority of a real-time task should be exposed in code and will be used as the priority value of the root in the concurrency tree. All sublevel families inherit priority from their immediate higher-level family, which in turn may inherit or be explicitly assigned if it is the root of the task. Such a foundation will instruct the Microgrid to conduct biased resource allocation as shown in Figure 1 and **Error! Reference source not found.**. Figure 1 shows a request for allocation of resources at a place (a cluster of one or more cores) where a minimal set of resources is reserved so that the family is run able. In principle, this will be guaranteed by the reservation of resources at a given priority level.

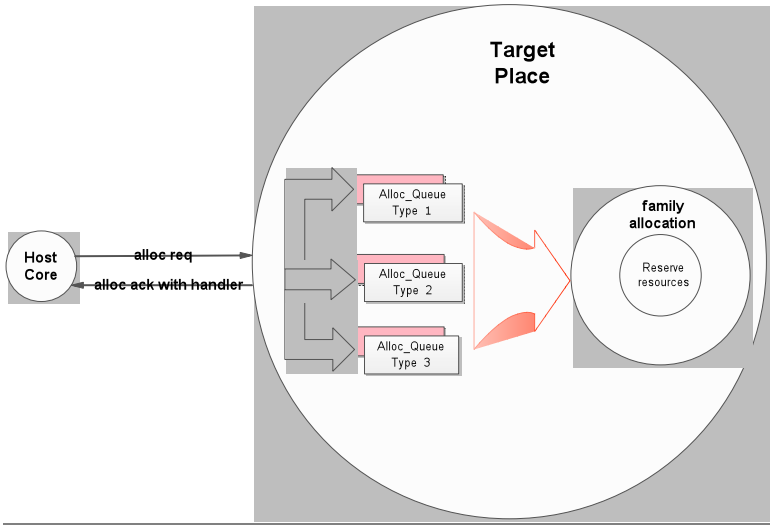


Figure 1. Family allocation: reserve minimum resources for a serial execution. It guarantees a runnable family on the Microgrid. N.b. in this and subsequent diagrams, multiple queues supporting different priority levels are identified by the red shadow to an existing queue.

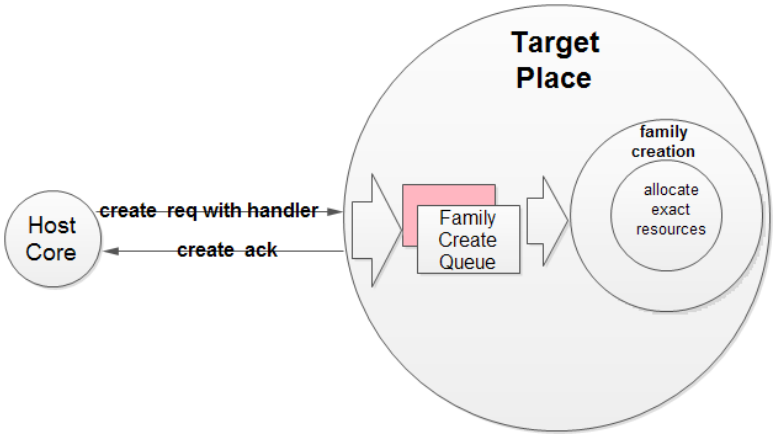


Figure 2. Prioritized family creation to get support of resources for fully execution.

Error! Reference source not found. shows the process of family creation, were unused resources are allocated to a family up to a predefined limit specified in the program.

The key new components implemented to achieve this biased resource allocation are the stand-alone buffers for requests to different priority levels. Processes responsible for allocating resources will scan these buffers from the highest priority to the lowest, pick out the first one to serve and then repeat the same routine. This promises the first-come-first-served allocation only to requests of equal importance and allows threads with higher priority come last and still be served first. A crucial issue here, especially in family allocation, is to ensure the availability of minimum resources whenever a real-time family request is received, so as to make it runnable. This is similar to the resource issues discussed in [7], which identified the policies as dynamic, dynamic with threshold and static. For the Microgrid, a hybrid solution is more flexible, that is to reserve resources for any forthcoming real-time task and restrain their usage by non-real-time families. How many resources to reserve, at any level of priority, is a matter of system-level design; it depends on the use cases for the system. For static priority, for example, it may depend on the number of potentially concurrent real-time tasks, whether the real-time task is able to use thread-level concurrency for speed etc. Where both apply it would be up to the system programmer to limit the resources allocated to one task to leave resources available for the other concurrent one.

With two levels of priority it is relatively simple to devise a resource reservation strategy. A high-priority task uses all resources and a low priority task uses only non-reserved resources. With many levels of the possibilities and resources required for efficient execution at each level are multiplied. We can turn to a customized dynamic allocation with threshold [7], although it may leads to performance loss, as there is no guarantee of acquiring resources for all levels. In this work we use only two levels of priority (low and high) and instead of reserving resources we use an upper bound on configured resources which is greater than those required by any of the experiments. This is possible in software emulation with the aim of determining what additional resources need to be implemented to guarantee available to different levels of priority for the applications tested in the target hardware.

A successfully created (mapped) hardware thread gets its unique context and is ready to be fetched into the pipeline as long as its first instruction is in the I-Cache. Once executed, it will be switched away on the following conditions:

1. *Current instruction is at the I-Cache line boundary.* On the Microgrid, a running thread is I-Cache miss free. As stated above, the prerequisite of entering pipeline is a full I-Cache line with instructions at the thread's PC. The boundary switch is forced to allow for the fetch of the next line just for this purpose. A thread that is swapped starts the process of filling another I-Cache line before waiting for a new opportunity to switch into pipeline again.
2. *Switch annotations.* These annotations are added by the compiler to the binary files on instructions dependent on prior high-latency instructions, also on branch instructions. This is an optimisation that allows instructions that may suspend on empty registers to be followed in the

pipeline by instructions from another thread. As there is no overhead for the context switch, this is preferable to flushing the pipe on such an occurrence.

3. *Missing data.* In the event that the compiler is unable to detect a dependency and a switch was not issued on an instruction missing data in one of its operands a switch will occur at the Register read stage.
4. *Miscellaneous.* This includes the end of all instructions in a thread, memory barriers, etc.

In order to ensure there are no I-cache misses we lock lines currently in use by one or more threads, n.b. the PC of an active thread defines the line in use. A count of the number of threads using each I-cache line is maintained and when a thread is made active, the appropriate count is incremented and when context switched, the count is decremented. Only when the count is zero will the line be released for re-use by other threads on an I-cache miss. With many threads each using unique I-cache lines, unnecessary context switches may introduce an impact on performance through I-cache line thrashing, which in turn can impact the real-time task performance even though resources for it are available. I-cache lines are shared resources and are not prioritised. It is imperative therefore that the I-cache is sized according to the characteristics of the thread load and expected sharing of the I-cache lines.

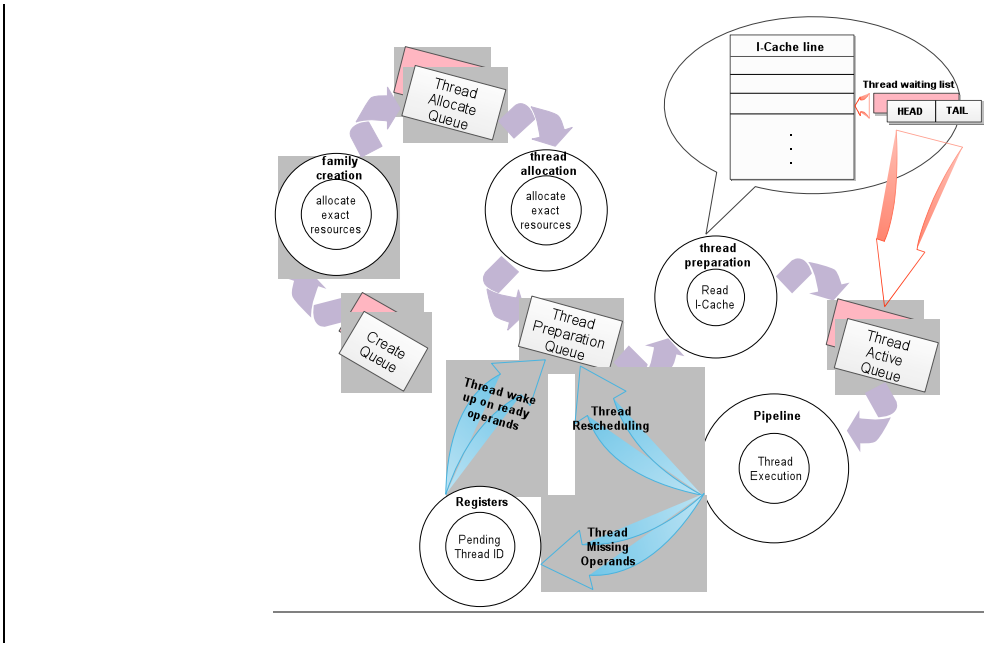


Figure 3. The activities of a hardware thread: creation, preparation, activation, execution and reschedule.

Figure 3 shows the various stages in the path leading to thread execution and the various buffers in that path. Once a thread has been allocated resources, a once off process, it may be switched many times. As explained above rescheduling a thread requires a guarantee that the thread's PC hits a line in I-cache. Given the frequency of this it is critical to ensure that this is not a bottleneck. It is also possible that threads of different priorities will read the same cache line, e.g.

calls to functions in the shared libraries. The process invoked on an I-cache miss forces any threads requiring that line to wait in list on the cache line and when that line arrives, these thread will be shifted to the active queue.

For priority scheduling we already require multiple active queues so threads can be selected on a first-come-first-served basis within a given priority but where this ordering may be reversed between threads of different priorities. However, the existing implementation moves all threads in a thread-neutral list on the I-cache to the thread-neutral active queue list and does so in a single cycle. N.b. a list only holds pointers to the head and tail of the entries in the thread table, where links define the list, hence appending a whole list to another is a two operation process. It updates one link field (the old tail's) and the tail field for the new list. Having one waiting list per cache line with mixed priorities complicates the shifting of threads from this list to the active queue. It is a stateful process that requires as many steps as there are threads in the lists; it may also invert the priority serving them on a first-come-first-served basis to their respective active queue lists. Consequently, each cache line must hold waiting lists for each level of priority, as with the active queues. In this way, all waiting lists can be appended to their corresponding active queues in a single cycle on the arrival of the cache line and the priority of the lists can be taken into account.

On the other hand, the suspension on registers behaves quite differently because each thread has its own register context and it is only global registers that may have multiple threads waiting on them, which by definition will all be of the same priority as only threads in the same family may read that family's global registers.

In addition to above switching scenarios, the priority scheme introduces a slightly different speculation on labelled context switches. Normally when the active queue is empty the currently running thread will not be switched as this switch is an optimisation and it may find data in its registers, so it is better to context switch when this speculation is proved false. Nothing is lost as there are no threads in the active queue to fill the pipe if switched. With priority queues the same action is applied, even though a queue of lower priority may not be empty.

Finally a mild form of pre-emption is applied to the active queues. As the fetch stage of pipeline always keeps an eye on the status of the active queues, whenever there is a non-empty queue with a higher priority, the currently running thread is forced to switch and the head thread in that queue is picked out to execute in the next cycle. Thus real-time threads always come to execution as soon as they are ready.

Putting all the above together, the static priority scheme equips the Microgrid with a basic capability to deal with real-time jobs and the fruit is shown in the experiments. However, many resources on chip, especially memories, are shared indicating the inevitable interference among threads with different priorities and the lack of predictability and analyzability of WCET, unless there is architectural supports, e.g. resource isolation or prioritisation.

3 Experimental Results for Real-time tasks in MGSim

3.1 Synthetic Benchmarks

In order to evaluate the benefits of implementing priority in MGSim, we have designed two benchmarks to represent the spectrum of anticipated real-time tasks. The first is the computation of square root, which is small, i.e. requires few processor cycles, sequential and computationally intensive, i.e. it does not have many memory operations compared to floating point operations. We believe this is characteristic of a number of iterative control algorithms used in space applications.

The second benchmark is a packet router task. This requires a significant number of processor cycles, is parallel and memory intensive as the major art of the benchmark is a parallel copy of a message in an input buffer to the addressed output buffer. We believe that it is characteristic of many communication algorithms used in space applications.

We implement square root by Newton iteration. It implements the following, where x converges to the sqrt(y)

xo:=x; x:= (xo+y/xo)/2.0;

The benchmark can be implemented with different precision, e.g. 1, 2, 4 and 8 decimal places. The algorithm converges quadratically, which is is equivalent to 2, 3, 5 and 9 iterations of the above equation for the single thread case. Although the algorithm is sequential algorithm we implement it with concurrent but dependent threads and investigate its behaviour with different numbers of threads exposed to the hardware.

The routing algorithm based is loosely on TCP/IP. It reads a header and extracts a 32 bit address which determines the output buffer required to route the packet input (1 of 4). It then reads the packet length and copies the packet from the input buffer to the addressed output buffer - we vary the packet size from 64 bytes, which is largely routing overhead, 4Kbytes, 16Kbytes and 64Kbytes. It is implemented with a parallel copy, i.e. each thread copies one byte from the input buffer to the output buffer.

3.2 Sparc vs Alpha ISAs

Known inefficiencies, e.g. unecessary pipeline stalls on the use of double loads plus a number of toolchain issues in compiling Sparc code meant the emulation of the Sparc ISA would not show what is capable from the microthreaded architecture. We therefore include here, results to demonstrate the implementation differences in micro-architecture and to show the results are qualitatively similar, although differ in peak performance between these ISAs due these issues.

We implemented both 32 bit (32K length) and 64 bit (64K length) FFTs on both Sparc and Alpha ISAs over a range of parameters (number of cores and number of threads used per core). As shown in Figure 4, which uses the 32 bit operands, the results are very similar for the two ISAs. The peak efficiency on the Sparc ISA is reduced by about 14% compared to Alpha due to differences in the ISA.

However, if we execute the FFT with 64-bit operands, as shown in Figure 5, we see that the efficiency of Sparc is only about 50% of that for the Alpha (i.e. one Alpha core is equivalent to two Sparc cores) due to the large number of double loads in the FFT, which stall the pipeline for one cycle per load. We note that the IPC of the 4-core Alpha execution is not twice that of the 2-core run, as it saturates the memory system.

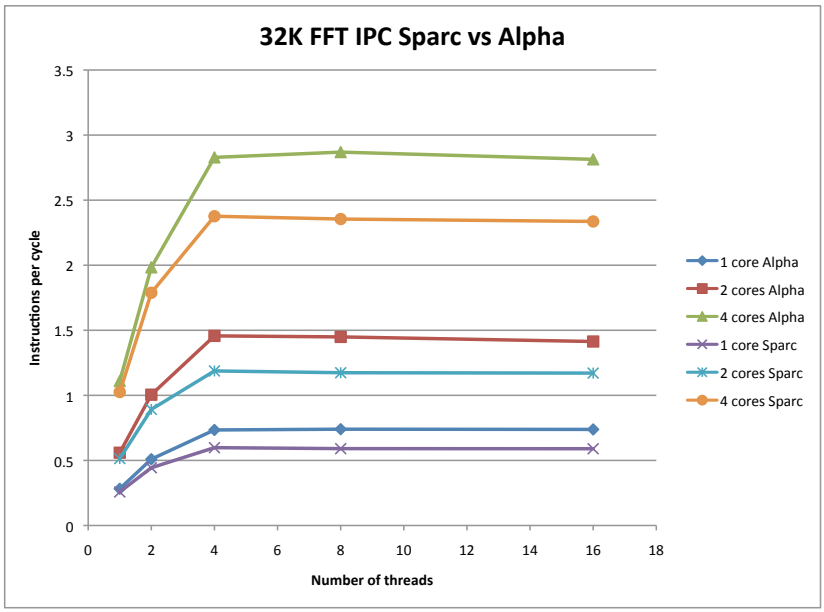


Figure 4. Results for 32K FFT using 32 bit operands comparing execution on Sparc and Alpha ISAs. The results show IPC.

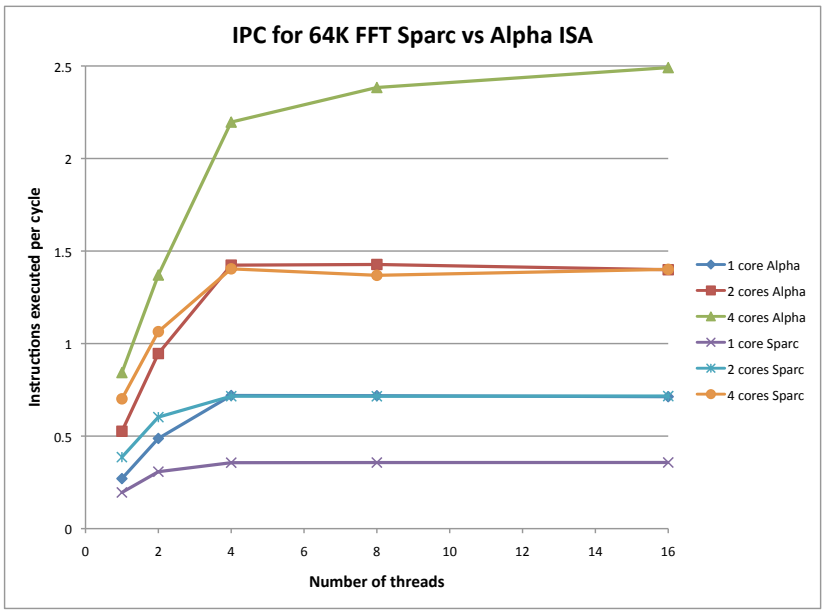


Figure 5. Results for 64K FFT using 64 bit operands comparing execution on Sparc and Alpha ISAs. The results show IPC.

Because of these results, we agreed to run all evaluations of the real-time thread benchmarks on the Alpha ISA.

3.3 Configuration of the Emulation platform for the Real-time experiments

All experiments presented here were implemented with the following emulation configuration:

- Four off 400MHz Alpha cores with microthreaded extensions;
- 2 off FPUs, shared between two cores;
- Pipelined I/O bus and on-chip memory as specified below:
    - 16K I-cache and 4K D-cache per core (4-way)
    - 1 off shared 256K L2 cache (4-way)
    - 2 off DDR2-800 channel
- On chip concurrent resources i.e. Thread and Family table sizes were as required by the benchmarks - we evaluate how many families, threads and what RF size would be required through evaluation on a cost performance basis
- The reservation of Priority thread slots, FT entries and registers is similarly as required.

3.4 Basic Scenario for the Real-time Experiments

In all experiments we run the real-time task against a long running background task. We have chosen the 64K FFT as the background task against which the RT benchmark is executed. The FFT is both computationally intensive as well as non-local in its memory requirements, so it exercises all parts of the system. In the algorithm, at first the data required is local and likely to be sourced from the local cache. However in the later stages of the algorithm, the data required will not be local and so will exercise both L2 cache and memory system.

Figure 6 again shows the results from running the FFT algorithm as a baseline, i.e. with no other tasks competing for resources. It shows speedup and average efficiency per core from 1 to 4 cores and from 1 to 8 threads. As can be seen, the best speedup is with 4 cores where the benefit of moving from 4 to 8 threads is negligible. With 4 cores and 4/8 threads the efficiency is limited by memory bandwidth at around 50% per core. For 1 and 2 cores efficiency is higher with a maximum between 80 and 90%. That is 90% of all pipeline cycles utilised. There is also a significant increase in efficiency between 4 and 8 threads (and hence speedup).

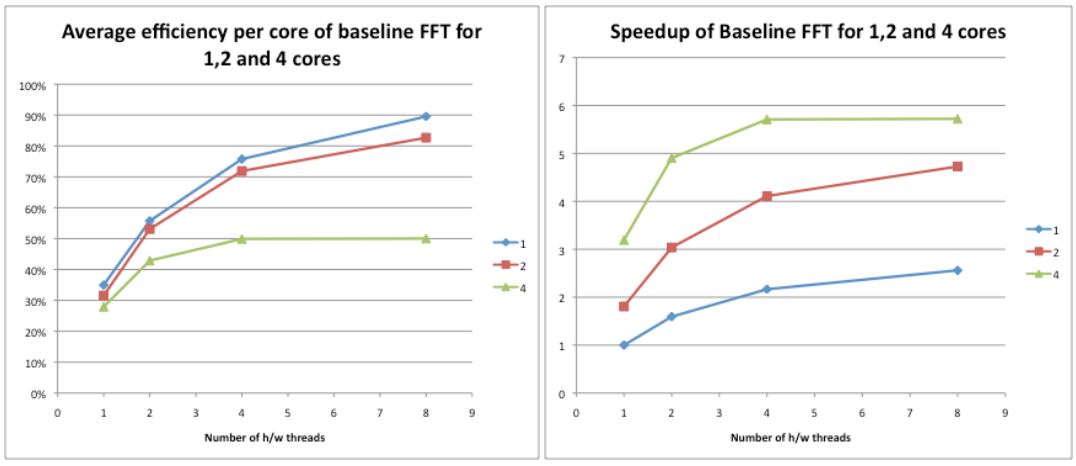


Figure 6. 64K FFT executing on the 4-core ESA configuration. On the left is average efficiency per core and on the right is speedup vs single core/single thread.

We have run two sets of experiments, the first execute a single instance of the real time task during the execution of the FFT. We do this sweeping through various parameters, namely data size for RT task, number of cores and threads for both RT task and background task. The RT task is executed well into the FFT at 100,000 cycles into the 10 to 50 million cycles taken by the FFT.

In the second set of experiments we execute the RT task periodically during the execution of the FFT for a limited range of realistic parameters.

3.5 Results for Parameter sweep on RT tasks

3.5.1 The Sqrt algorithm - sequential, computationally intensive RT task

Figure 7 shows the execution time of the Sqrt algorithm on a single core for both 1 and 2 threads. It can be seen that the 2-thread version is slower than that for a single thread. However it executes more instructions, both in setting up the task and in executing it. The linear interpolation of the results shows that the overhead for setting up and synchronising the task is a little below 200 and 250 cycles respectively for 1 and 2 threads. However, the 2-thread version is slightly faster per iteration. As expected, if we look at efficiency (see Figure 8) the 2-thread implementation shows a small increase in efficiency but as the algorithm is sequential we get only ~30% efficiency in both cases. We restrict the remaining experiments for Sqrt therefore to the 1-thead version.

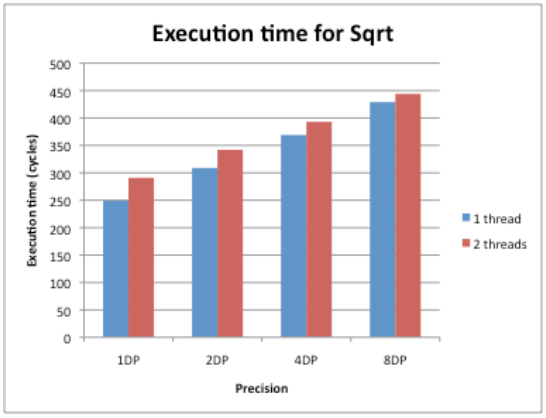


Figure 7. Execution time of Sqrt task for different precisions for 1 and 2 threads.

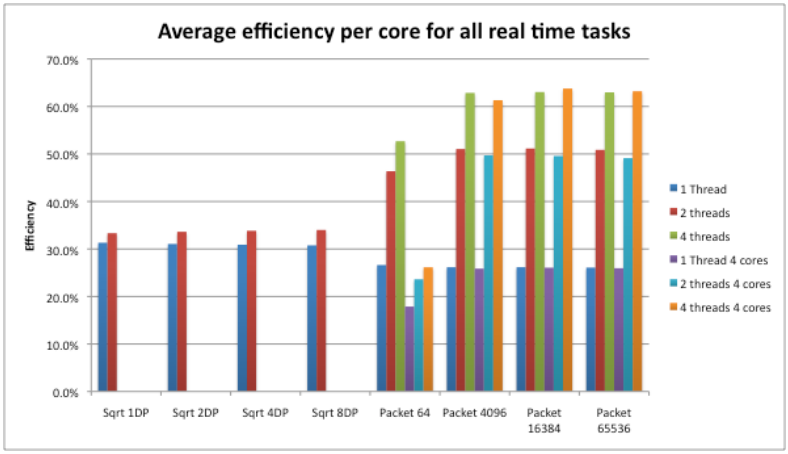


Figure 8. Efficiency of all RT benchmarks across all threads and cores used in the experiments.

The above results represent the baseline for comparison in the following experiment. We look at the execution time of the same algorithm, limited to one thread and compare the execution time to the baseline above. In order to investigate the impact of different activity sharing and FFT parameters we define the following experiments.

1. *Base* - the RT thread is executed stand alone on one core results above.
2. *Shared L2 Cache* - the FFT is executed on a 2-core place and the RT thread is executed on one of the remaining cores, which does not share an FPU with FFT. In this way we isolate the sharing of a core or FPU unit (shared between 2 cores). However, this is unlikely to place a maximum load on the L2 cache as we see a peak efficiency of over 80% in the FFT with 2 cores and 8 threads. This shows that the FFT is able to tolerate memory operations and the bandwidth limit on memory is not exceeded.
3. *Shared FPU* - the FFT is executed on one core and the RT thread is executed on the core sharing the same FPU.
4. *Shared core* - the FFT is executed on all 4 cores and the RT thread is executed on one of the cores at the same priority as the FFT.
5. *Shared core P* - as above but the RT task is executed using a priority thread.

We measure the impact of sharing resources (Space sharing) in experiments 2 and 3 and in using priority by comparing the execution time of experiments 4 and 5. Experiments 4 and 5 are then repeated with the FFT running on a single core.

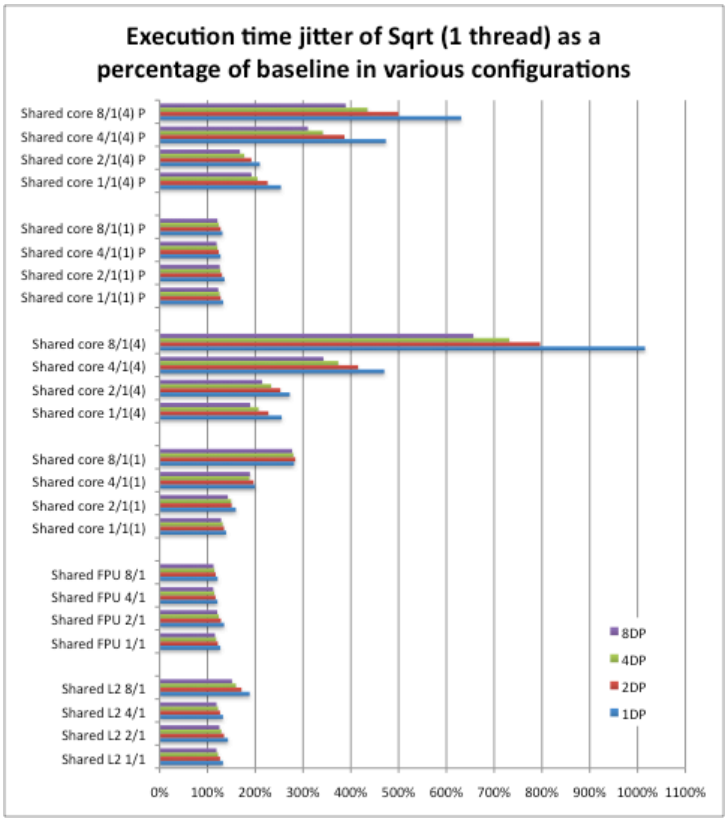


Figure 9. Execution time jitter for Sqrt across a range of parameters and sharing conditions plotted as a percentage of the baseline execution time. In all labels the parameters x/y represent the

hardware threads allocated to FFT and Sqrt respectively and the figure in brackets for the shared core experiment indicates the number of cores on which the FFT runs.

Figure 9 shows the results of all experiments in the parameter sweep experiments. Each run was compared to the Sqrt function executing stand alone and plotted as a percentage of this baseline execution time. It should be noted that only the shared core experiments use priority threads. The other two (shared L2 and shared FPU) are included to illustrate the impact of sharing resources when space sharing the real-time task.

It can be seen that the impact of sharing FPU is low. The worst case is a 35% increase in execution time for the lowest precision result and for the full precision is no more than 20% overhead. This shows that the FPU is well provisioned.

There is a larger impact in sharing the L2 cache. The worst case is an 88% increase in execution time for the lowest precision result and for the full precision is no more than 51% overhead. This shows that, for two cores at least, the L2 cache is also well provisioned. Note that the largest overhead is for 8 threads of FFT, where the FFT is running at more than 80% efficiency. Note that even though the Sqrt task is not memory intensive, the creation of threads and their execution requires memory operations to set up the family parameters and fetch cache lines.

For the shared core experiments, we see a similar trend in all results. The overhead increases with increasing FFT threads (i.e. more efficient FFT execution) and decreases with precision, confirming the impact is in the set up of the Sqrt execution. For a single core without priority the worst-case overhead is 184%, which is reduced to 36% when using priority. For the four-core case the impact is much worse. Without priority, in the worst case, there is a 10 times increase in execution time compared to the base case, which is reduced with priority to a factor of 6. In both cases this is for 8 FFT threads and 1DP accuracy, i.e. a very short task subject to significant overheads in setup costs, probably due to contention on memory.

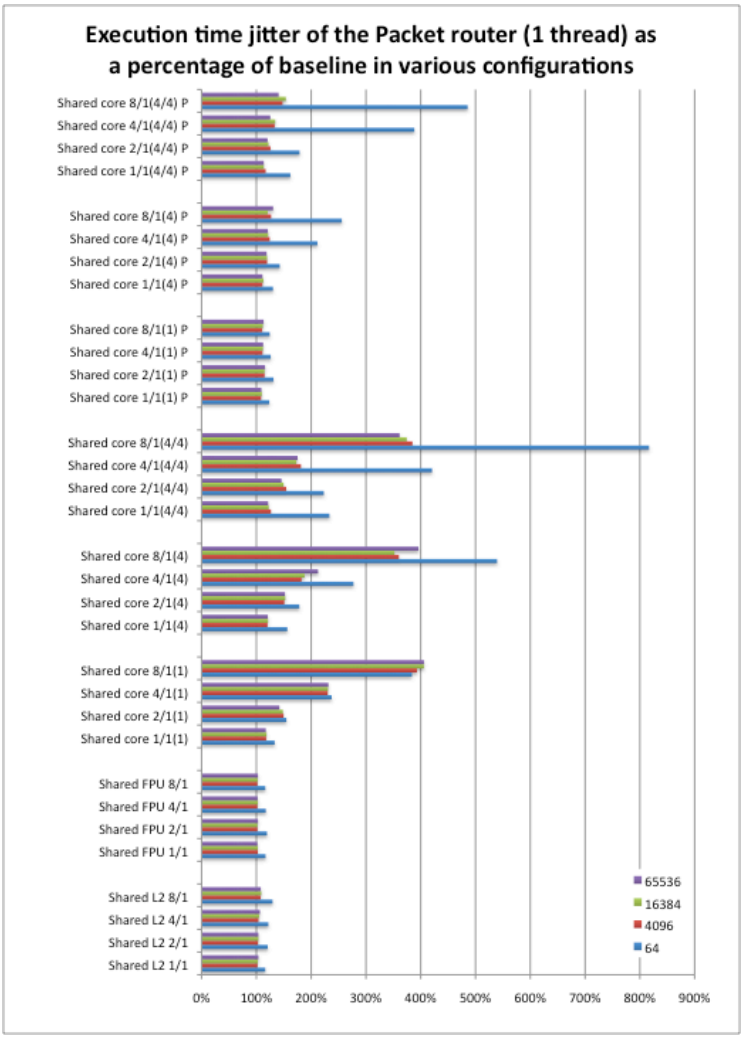


Figure 10. Execution time jitter for the 1-thread Packet router across a range of parameters and sharing conditions plotted as a percentage of the baseline execution time. In all labels the parameters x/y represent the hardware threads allocated to FFT and router respectively and the figure in brackets for the shared core experiment indicates the number of cores on which the FFT runs, in one experiment (4/4) the Packet router is also run on 4 cores.

3.5.2 THE Packet Router Algorithm - large, parallel, memory intensive task

Figure 10 to Figure 12 show the results for all experiments in the parameter sweep experiments for the Packet router algorithm. Each run was compared to the same function when executing stand-alone and is plotted as a percentage of this baseline execution time. In this case, because there are performance gains to be made, the results are shown for 1-, 2- and 4-thread executions. Again it should be noted that only the shared core experiments use priority threads. The other two (shared L2 and shared FPU) are included to illustrate the impact of sharing resources when space sharing the real-time task.

In all of these plots we see a similar pattern to that seen in Figure 9. For the space-sharing experiments, there is very little additional execution time, while sharing resources with the FFT background task. A 30% overhead for the smallest packet size is the worst case for a single thread and excluding the smallest task, the maximum overhead is only 9%. For two threads this increases slightly to a maximum of 58% for the smallest task and 24% otherwise. For 4 threads there the figures are 45% and 29% respectively. As to be expected, the

worst case execution times are for the shared L2 cache experiment, as the packet router is not a computationally intensive kernel but memory bound.

Considering the shared core experiments on a case-by-case basis, for a single core, labelled (1), we see the most predictable results. Without priority the overhead on the real-time task increases with the number of FFT threads resulting in approximately 4 times the baseline for 1 and 2 RT threads and

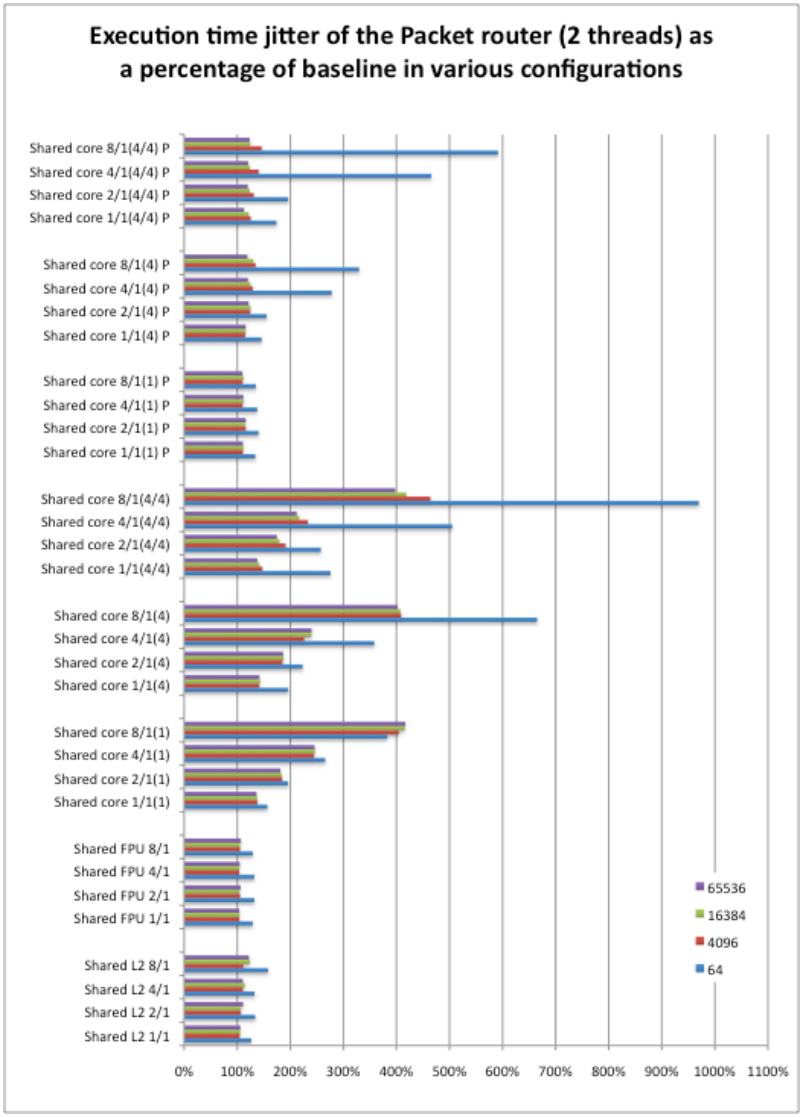


Figure 11. Execution time jitter for the 2-thread Packet router across a range of parameters and sharing conditions plotted as a percentage of the baseline execution time. Labels as Figure 10.

decreasing to about 3 times for the 4-thread packet router. We believe this result illustrates the fair scheduling in the Microgrid core, i.e. the more threads used for the RT task, the greater the share of resources the task uses and hence the overhead of running against the background task is reduced. Note that this experiment is not bandwidth limited as it uses just a single core.

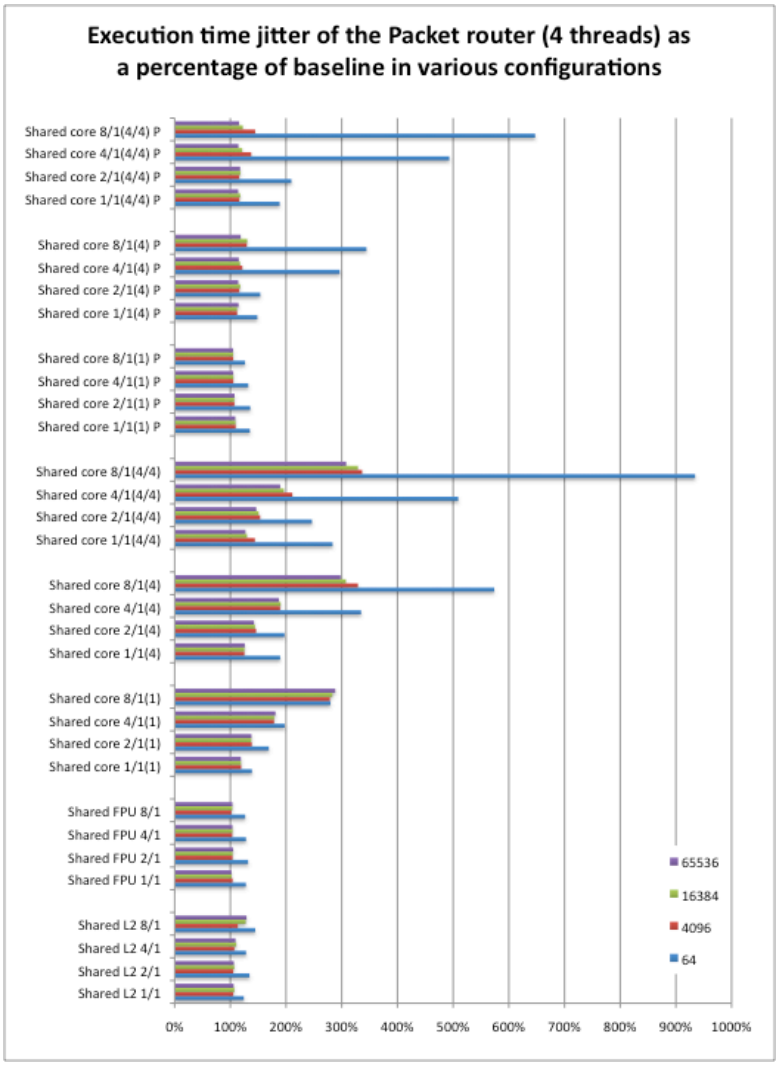


Figure 12. Execution time jitter for the 4-thread Packet router across a range of parameters and sharing conditions plotted as a percentage of the baseline execution time. Labels as Figure 10.

If we compare this with the RT task running with priority threads the difference is significant. The maximum overhead is 40% across all parameters and this reduces to 16% if we exclude the smallest packet size. These are very predictable results.

For both of the experiments running FFT on 4 cores, the results are similar to the single core, except for the smallest packet size. Without priority, there is a gradual increase in execution time with FFT threads up to about 4 times the base execution time, whereas for the smallest packet this increases to around 10 times. The priority thread results are constrained to within an additional 50% for the longer packet sizes but for the smallest packet size, we also see a large increase in execution time of up to 6 times the base case, which increases with the number of threads executing. The fact that this overhead reduces with packet size is indicative of a one off delay in setting up the algorithm, i.e. in the process of creating threads. This is quite likely caused by memory accesses delayed because of bandwidth limitations when running relatively large numbers of threads on all four cores. We see this in the reduction in FFT efficiency when moving from 2 to 4 cores, see Figure 6.

3.6 Results for Periodic RT Tasks

For the second set of experiments we wanted to look at a realistic execution schedule for the real time task and look at the average execution time statistics and jitter over the set of results. To achieve this we selected typical parameters for the two RT tasks, namely 8DP for Sqrt using a single thread (~600cycles for the base case) and a 16KByte packet for the packet router using 4 threads (~10,000 cycles for the base case). These RT tasks were executed periodically while concurrently executing a 64K FFT running with 8 threads. Again we compare the statistics with and without priority threads for the real-time task. The configurations used were as follows:

1. FFT and RT tasks both executed on a single core (1+1);
2. FFT executed on 4 cores RT task executed on 1 core (4+1);
3. FFT executed on 4 cores RT task executed on 4 cores (4+4) - packet router only.

The real-time tasks were implemented in a loop using a delay between executions and start time and execution time were recorded for each execution. Similar statistics were recorded for the FFT running in a separate independent thread. We present results for only those RT tasks executing during the FFT execution.

3.6.1 The Sqrt Algorithm

The Sqrt task was executed at a nominal rate of once every 10,000 cycles and again at once every 20,000 cycles, which with timing overhead is a nominal duty cycle of 10% and 5% respectively for the RT task. When running concurrently with the FFT, an overhead in the timing threads increased the period measured, which varied between 1200 for both threads on the single core and on 4 cores with priority thread to around 1800 for non priority on 4 cores. For the 2,00 cycle nominal case the corresponding figures were 2200 and 2800. Note that both this variation in periodicity and the variation in run-time of the FFT had an impact on the number of RT tasks executed in the different experiments.

We first present the variation in execution time of the background FFT task, which is shown in Figure 13. For the 1-core experiments, the FFT is executing at about 90% efficiency (see Figure 6). Without priority, if you consider the additional instructions that have been executed in the RT task, there is little impact on the performance of FFT. There is more effect when using priority but still the impact is relatively minor being only about a 5% reduction in execution efficiency.

For the 4-core experiments we see an interesting effect. Without priority, the result of running the RT task is to speed up the execution of the FFT. We know the FFT is running at low efficiency because of memory pressure and we have observed that modifying the schedule of memory requests can have an impact on performance, i.e. it is not the average request rate but peak rate than can limit performance. Again performance is affected negatively with priority but considering the number of additional instructions executed per core it is only the (4+4) case where there is any appreciable overhead, where in this case it gives a 15% reduction in execution efficiency.

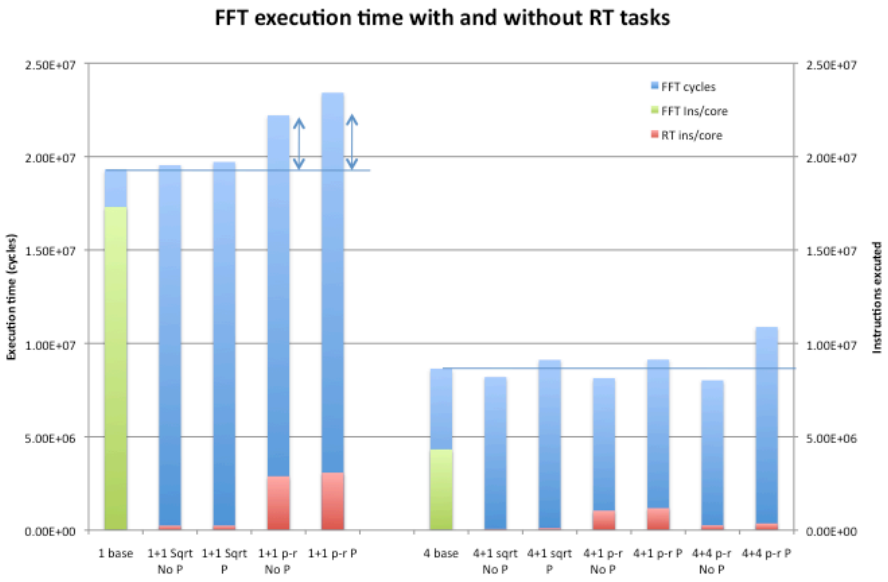


Figure 13. Execution times of 64K FFT running with and without periodic real-time time tasks. The blue bars show the execution times. The green overlay on the baseline execution time is the number of instructions executed by the FFT. The red overlay represents the additional instructions executed by the real time task during the FFT's execution.

Figure 14 shows the jitter in execution time for all iterations of the Sqrt RT task. It plots the execution time for each iteration. It can be seen that using priority (red and blue tracks) reduces both execution time and jitter in the execution time compared to the same experiment without priority. The other clear result is that jitter is significantly less when running the FFT on a single core. The reason for this is quite obvious, running FFT on 4 cores puts a much larger pressure on the shared resources, in particular the memory system, which is not prioritised and can cause significant latency if request messages get queued. The load the FFT puts on the memory system is not constant over its execution and this can be seen as well. The communication pattern of FFT is non-local. Data is required at stage k = 0..n-1, for a 2^n transform, from a skip distance of 2^k relative to the local writes, so that there will be increasing bus activity as k increases and the effects of locality diminish. Although Sqrt is a computationally intensive task and hence does not require many memory operations, it has a relatively small execution time. Hence the initial memory reads and final store of the result could add significantly to its execution time if the memory requests are being queued. There could also be delay in memory operations involved in creating the task itself.

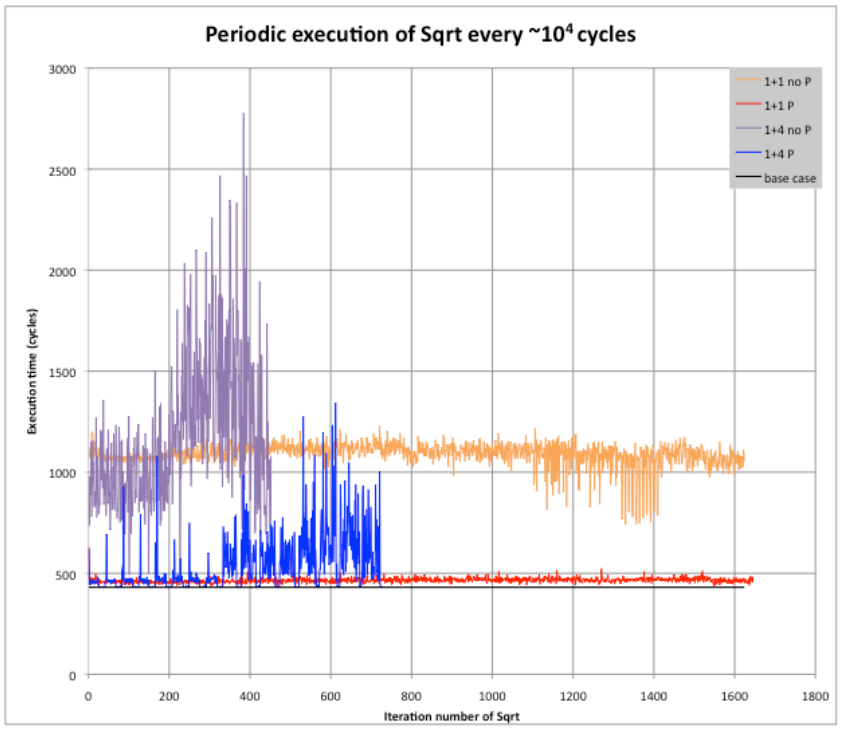


Figure 14. ***Execution time of multiple Sqrt tasks nominally every 10,000 cycles, executing concurrently with FFT on 1 and 4 cores.***

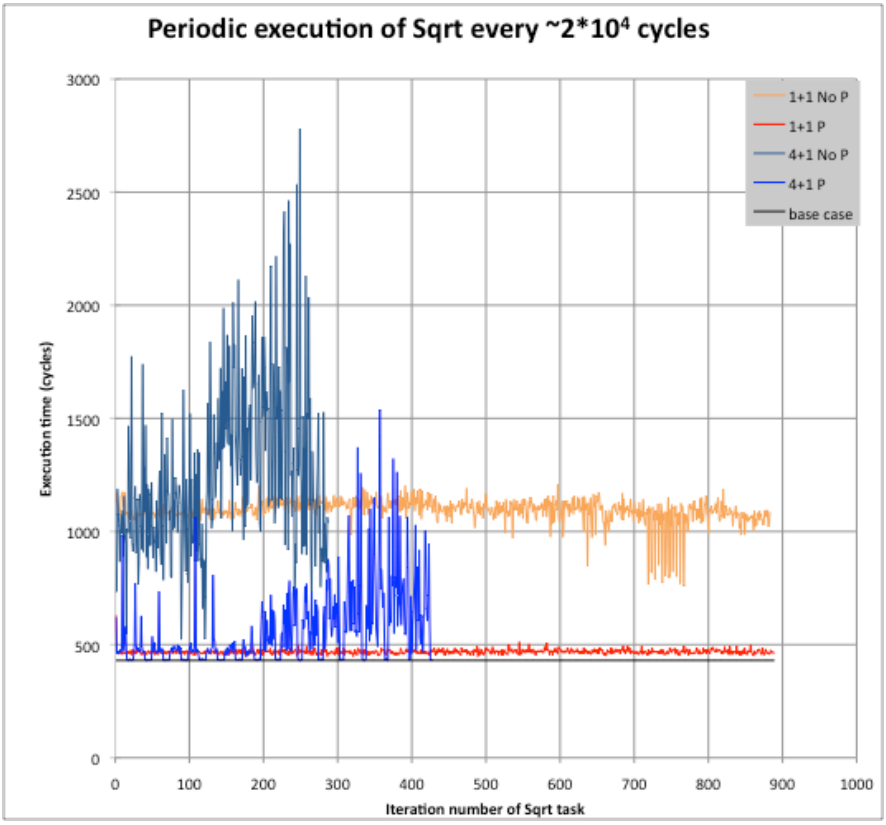


Figure 15. ***Execution time of multiple Sqrt tasks nominally every 20,000 cycles, executing concurrently with FFT on 1 and 4 cores.***

On a single core, the impact of memory contention is minimal and we see a very consistent execution time, which is some 5-10% above the base case. Even without priority execution time on a single core is still very consistent but using priority gets the job done some 2.34 times faster on average with an absolute jitter reduction of 2.52. In both cases the jitter as a percentage of average execution time is 42% with priority and 45% without.

On 4 cores, the average speedup using priority is 2.19 times and the jitter is reduced in absolute terms by 1.56 times. Here the jitter as a percentage of average execution time is 171% using priority and 199% without.

	Maximum	Average	Minimum	Base case
1+1 No P (10^4)	1228	1090	740	430
1+1 P (10^4)	631	465	437	430
4+1 No P (10^4)	2776	1172	442	430
4+1 P (10^4)	1343	534	431	430
1+1 No P (2*10^4)	1208	1093	758	430
1+1 P (2*10^4)	631	468	448	430
4+1 No P (2*10^4)	2778	1253	523	430
4+1 P (2*10^4)	1538	560	431	430

Table 1. ***Execution time statistics for Sqrt with and without priority on 1 and 4 cores. The figures in brackets show the nominal cycle time of RT task execution.***

Figure 15 shows the results for the situation where the load imposed by the RT task is halved. The results are qualitatively very similar with the overall pattern being reproduced with approximately half the number of executions.

In this case, on a single core, using priority gets the job done some 2.33 times faster on average with an absolute jitter reduction of 2.46. In both cases the jitter as a percentage of average execution time is 39% with priority and 41% without.

On 4 cores, the average speedup using priority is 2.23 times and the jitter is reduced in absolute terms by 2.03 times. Here the jitter as a percentage of average execution time is 180% using priority and 198% without.

Table 1 and Figure 16 show the aggregated statistics for the Sqrt RT task for both frequencies of execution. We define jitter as reported above as the difference between maximum and minimum values in this table. For information and further analysis, the interested reader is referred to the source data in the spreadsheet submitted along with this report.

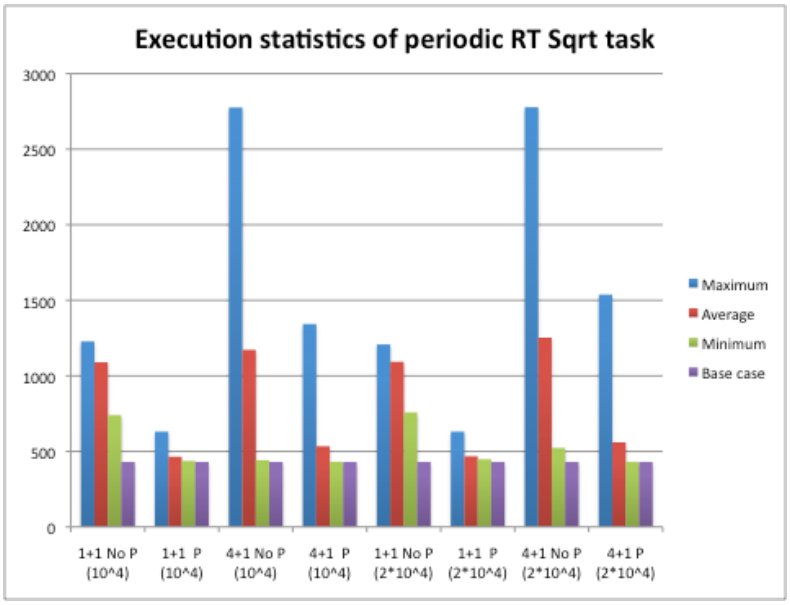


Figure 16. ***Statistics for Sqrt executions with and without priority on 1 and 4 cores. The figures in brackets show the nominal cycle time of RT task execution.***

3.6.2 The Packet Router Algorithm

The packet router was executed nominally every 0.5 for the high duty cycle and 1.0 million cycles for the low duty cycle experiments. The baseline execution times for the16K packet router task running stand-alone are given in Table 2. The first thing to note is that because of the good locality properties of this code we see almost ideal speedup, 3.98, once the caches are warm. We also note that the duty cycle of these experiments is around 20% and 10% respectively. As with the previous experiment, concurrent threads manage and time the execution of all the packet router tasks in one and the execution time of the 64K FFT background task in the other.

	Solo-core	Quad-core	Speedup
Cold base	105112	26594	3.95
Warm base	102457	25701	3.98

Table 2. Baseline execution time in cycles for the 16K packet router task on one and four cores.

It can be seen from Figure 8 that the baseline efficiency (per core) of the packet router is very similar for both single and four-core cases. It starts at 25% for a single thread rising to 50% for 2 and 60% for 4. We also note from Figure 6 that on 4 cores the FFT executes at an average of 50% efficiency. In these experiments we use 4 threads to implement the packet router so clearly there is the potential for more impact on performance than with the Sqrt kernel, which only executes at 30% efficiency.

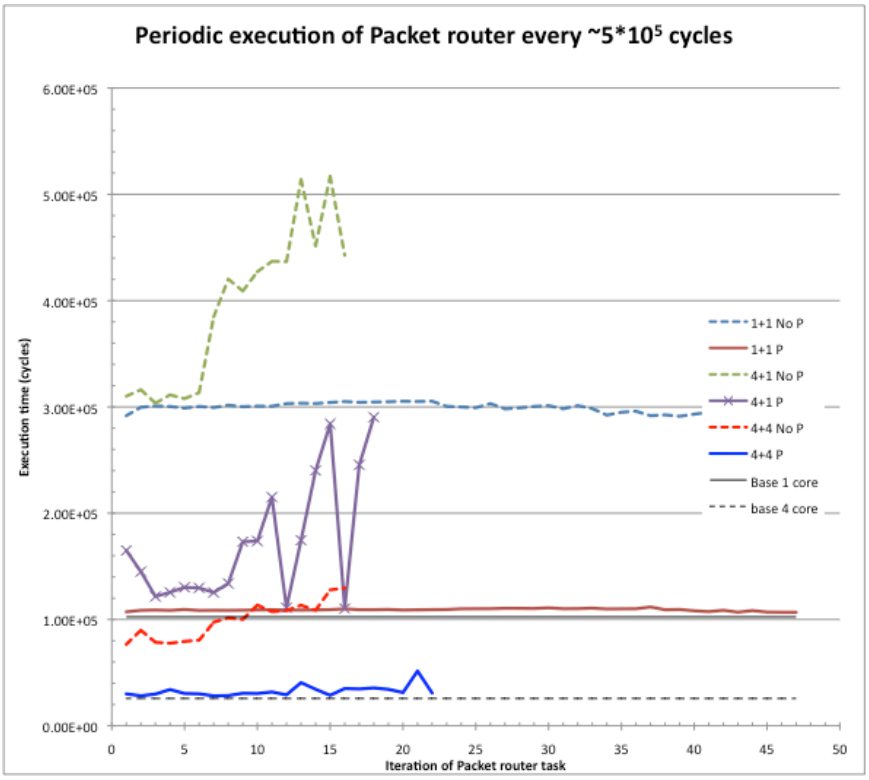


Figure 17. ***Execution time of multiple Packet router tasks nominally every half million cycles, executing concurrently with FFT. Both FFT and RT tasks are executed on 1 and 4 cores with the mixed case 4+1 being 4 cores for FFT.***

Figure 17 shows the results of the high duty cycle experiment. The first key result we notice is that running the priority task over the same cores as the background task reduces execution jitter. This is true for the single core case where the memory system is not saturated but also we note for the case where both FFT and Packet router run on all four cores. Whereas in the (4+1) experiment, we see that even with priority the execution time of the Packet router increases significantly as we progress through the different stages of the FFT, which as already explained give an increasing load on the memory system due to non-local communication. This is not seen at all in the (1+1) experiment and is substantially attenuated in the (4+4) experiment, especially when using priority.

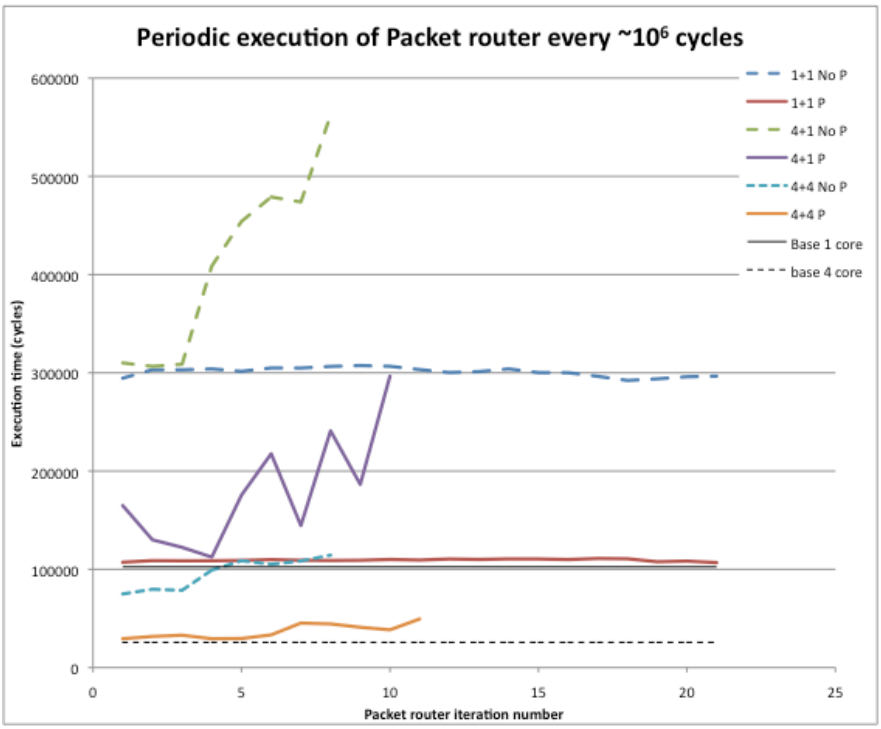


Figure 18. ***Execution time of multiple Packet router tasks nominally every million cycles, executing concurrently with FFT. Both FFT and RT tasks are executed on 1 and 4 cores with the mixed case 4+1 being 4 cores for FFT.***

As before, we see a consistent speedup in execution time between using priority threads and not. For the single core case, the average period is $5.02*10^5$ cycles both with and without priority, so the timing overheads are not so significant in this longer period. The speedup using priority is 2.74 and the reduction in jitter is 2.84. The absolute jitter is very low, just 4.7% and 4.5% of the average execution time for no priority and priority respectively.

For the (4+4) case, the period is 5.11 and $5.04*10^5$ cycles for no priority and priority respectively. The speedup using priority is 3.05 and the reduction in jitter is 2.27. The absolute jitter is higher in this case at 53% and 71% of the average execution time for no priority and priority respectively.

For the (4+1) case the actual period averages out at 5.13 and $5.05*10^6$ for no priority and priority respectively. It is in this experiment that we see the worst jitter. This is not surprising as we have three cores running FFT only, which unlike the (4+4) case, are un-moderated by running priority threads. This means they continue to place a large load on the memory system, while the priority threads are running on the fourth core. With this case, the speedup using priority is 2.29 and the reduction in jitter is only 1.19. The absolute jitter here is 54% and 104% of the average execution time for no priority and priority respectively, confirming the impact on priority threads from this un-moderated load on the memory from the other three cores.

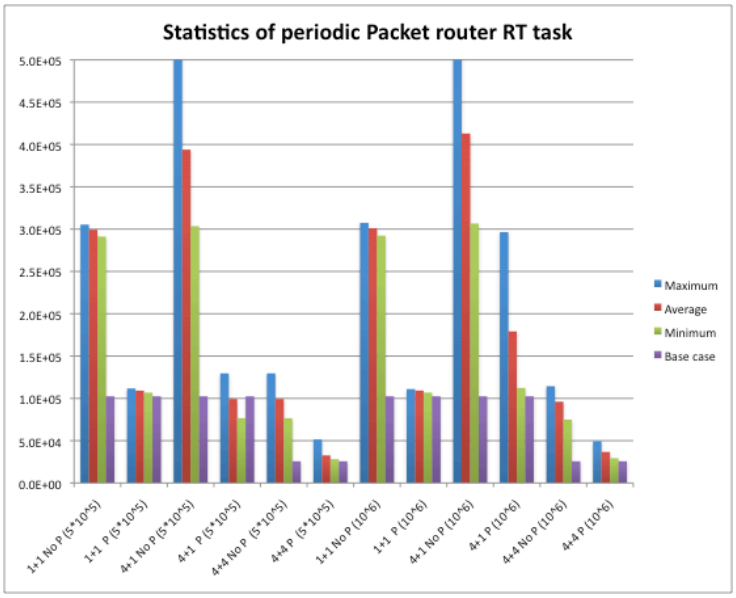


Figure 19. ***Statistics for the Packet router executions with and without priority on 1 and 4 cores, including the execution of the RT task on 4 cores. The figures in brackets show the nominal cycle time of RT task execution.***

Figure 18 shows the results for the low-duty cycle case, with nominal period of 10^6 cycles. We see a very similar picture with the (1+1) and (4+4) cases giving low jitter and the (4+1) case showing the same increase in execution time as the FFT task progresses. The aggregated results for this duty cycle are both qualitatively as well as quantitatively similar to the higher duty cycle experiment, with the exception of the fewer RT task executions.

	Maximum	Average	Minimum	Base case
1+1 No P (5*10^5)	305360	299269	291206	102457
1+1 P (5*10^5)	111752	109151	106781	102457
4+1 No P (5*10^5)	517017	393888	303486	102457
4+1 P (5*10^5)	129579	99393	76614	102457
4+4 No P (5*10^5)	129579	99393	76614	25701
4+4 P (5*10^5)	51507	32630	28169	25701
1+1 No P (10^6)	307421	301010	292254	102457
1+1 P (10^6)	111040	109253	106820	102457
4+1 No P (10^6)	562281	412938	306603	102457
4+1 P (10^6)	296530	179131	112415	102457
4+4 No P (10^6)	114431	96131	74900	25701
4+4 P (10^6)	49435	36773	29338	25701

Table 3. ***Statistics for the Packet router executions with and without priority on 1 and 4 cores, including the execution of the RT task on 4 cores. The figures in brackets show the nominal cycle time of RT task execution.***

Table 3 and Figure 19 show the aggregated statistics for the packet router experiments. What is significant to notice is the clustering of max, average and minimum execution times in close proximity to the base case in the (4+4) and especially the (1+1) experiments.

3.7 Conclusions on Priority Threads

Simulation of RT threads was performed using two synthetic benchmarks, one small and computationally intensive and the other larger and memory intensive. These are representative of control and communications algorithms respectively. The benchmarks were executed while a background task (a large FFT) was running. Various configurations were evaluated including tests to evaluate the use of shared resources when the RT thread executed stand-alone on a single core but shared resources with the background task. In addition to this experiments wee undertaken to run the benchmarks periodically and to collect statistics on their execution. This included running both the background task and the RT task (where possible) on both a single core and on all 4 cores.

For both benchmarks the following general results were observed:

1. Running the RT benchmark with priority threads always improved both execution speed and jitter, typically by between a factor of between 2 and 4 times for both.
2. When the background task was run on 4 cores it was clear that there was interference from the background task through use of non-prioritised shared resources (e.g. floating-point unit and particularly the L2 cache and memory interface), however despite this, using priority threads for the RT task always improved the RT task performance and jitter.
3. When both RT and background task were restricted to a single core, the execution time of the RT task was only marginally slower than its execution stand-alone and the jitter in execution time was negligible.

In conclusion, using dual thread priority in the Microgrid core gave significant improvements in both run time and jitter, however when background tasks placed stress on the memory system the benefits were not so obvious. Even so there was always at least a halving of the RT task execution time (both maximum and average over multiple executions) and jitter, when using priority threads over non-priority threads. In addition there was no significant impact on the execution time of the background task and in some cases the background task execution was improved, even though the number of instructions executed increased.

4 Thread-level Redundancy for Fault Tolerance

The vulnerability of multi-core processors is increasing due to tighter design margins and greater susceptibility to interference. Moreover, concurrent programming environments are the norm in the exploitation of multi-core systems. In this paper, we present an on-demand, thread-level, fault detection mechanism for multi-cores. On-demand redundancy allows users to set the redundancy scope in the concurrent code and is important to be able to manage the overheads of redundant execution; not all code is sensitive to errors. To achieve this we introduce a mechanism to provide redundant thread creation and synchronization, which manages concurrency and synchronization between the redundant threads via the master. This framework was implemented in the Microgrid emulation, which is a multi-threaded, many-core processor with single, in-order issue cores. It was evaluated using a range of programs in image and signal processing, and encryption. Preliminary results show the performance overhead for using redundancy in a kernel is less than 11% for single core execution and is always less than 100% for all scenarios. This efficiency derives from the platform's hardware concurrency management and latency tolerance.

4.1 Background and Motivation

With multi-core systems now being mainstream these days, the number of cores integrated into a processor will increase due to the inevitable technological progress. However, the same trend in technology also make the future of multi-core processors increasingly susceptible to both hard and soft errors. The shrinking of feature size leads to more manufacturing defects, process variations, and early lifetime failures [10]. Also, the reduction of design margins and the transistor's threshold voltage can increase the soft error rate of certain noise environments dramatically [11].

Redundancy is a classic solution for tolerating faults [12]. The key practical issue is how and at what level to apply redundancy in multi-core processors. Spatial redundancy such as IBM's S/390 G5 processor [13], Tandem S2 [14], Hewlett Packard's NonStop Advanced Architecture [15] and Boeing 777's flight computer [16] is not the most efficient approach, due to its large area and energy overhead. Also, this approach lacks flexibility when redundancy is not necessary. In contrast, temporal redundancy such as AR-SMT [17], SRT [18], SRTR [19] and BlackJack [20] has its advantages in area overhead and flexibility but is limited by the duration of error. Typically these approaches cannot detect hard or soft errors lasting longer than the interval of two thread copies.

Although multi-core processors have a higher vulnerability, they also provide natural extra hardware for fault tolerance. So multi-core processors based thread-level redundancy (TLR) techniques such as DCC [10], CRT [21], CRTR [22] and Reunion [23], which force two copies of a semantic thread to run on different cores, combine the fault coverage of spatial redundancy with the efficiency of temporal redundancy. They can detect both hard and soft errors without adding extra hardware. However, these TLR techniques focus on single-threaded environments. Much less attention has been paid to the design issues in

concurrent programming environments; it is not a trivial extension to migrate TLR from a single-threaded environment to a concurrent programming environment. In particular, concurrent software already uses multiple cores simultaneously for performance issues, and hence TLR must manage simultaneous occupation of the cores by the concurrent software and its replicates.

Meanwhile, many emerging applications allow for the discarding of individual sub-computations with small qualitative impact [24]. This is a form of *intrinsic* fault tolerance, which does not require extra architectural support. Also, many commodity systems do not need high reliability and some can tolerant faults to some extent, e.g. the odd pixel in video decompression cannot be noticed. Even for mission critical systems, fault tolerance is not needed all the time, especially when performance and energy are key considerations. In other words, most applications do not require an investment in the cost of full redundancy. And the most elegant solution is that fault tolerance is provided only when necessary, i.e. *redundancy on demand*.

To address these two problems, this section of the report presents an on-demand, thread-level, fault-detection framework in a concurrent programming environment. This vertical framework includes support from the programming model, compiler, ISA and micro-architecture, but none of the changes are on the critical path of the system. In other words, the fault tolerance mechanism is independent of the existing system. As with the other TLR mechanisms in multi-core processors, we propose two thread copies of a semantic thread, which are called the master and redundant thread respectively. They are forced to run on a fixed pair of adjacent cores in order to detect both hard and soft errors. The sphere of replication [18] includes the entire pipeline, the register file and L1 cache of each core. An assumption is made that the memory system is fault-free. Also, we adopt a relaxed input replication technique, so as to avoid significant changes to the existing cache hierarchies for redundant execution. The divergence of any load value, induced by relaxed input replication, can be corrected or recovered by the same mechanism employed for fault detection [23]. Finally, the output of the two thread copies is compared to check whether it is correct. Here we focus on the output to memory (i.e. *store*s) only, I/O operations have not been considered at this stage but we note that memory-mapped I/O would be covered by this scheme except where DMA controlled.

In summary then, the issues covered by this report are when, where and particularly how a program should be duplicated to give high reliability. The user or the run-time environment can specify when to protect against faults. Where and how the duplicates are executed is the main technical contribution of this research. Together, this makes the system more efficient and flexible as the granularity of redundancy is a thread, which can be specified anywhere in the hierarchical, threaded programming environment.

The main technique introduced is an intelligent redundant thread creation and synchronization scheme, which adds as little as possible to the concurrency management to automatically create and manage the synchronisation of both the master and redundant threads automatically. When using a concurrent

programming environment, we usually want to exploit it as much as possible. Because of this hierarchical concurrency and thread independence are two features of many concurrent-programming environments that affect the design of a thread duplication scheme. In our scheme, we only allow the master thread to create both master and redundant threads' child threads in order to avoid a thread explosion in the hierarchy. Additionally the master thread maintains synchronization between the redundant parent and its child threads. This requires some changes to existing systems and in the case of the Microgrid a co-design of hardware and system software.

4.2 Concurrent programming environment

The key feature of a good concurrent programming environments is to define concurrency in such a manner that it can be easily and efficiently mapped onto the available resources for execution, threads, cores etc., either statically or preferably dynamically, such that the mapping is not defined in the program. The latter provides for a more portable code and requires concurrent activities be scheduled onto the same resource.

Generally, such systems can be modelled as a fork-join queue within a closed queuing network. Each incoming workload is split into N tasks at the fork point, and each of these tasks queues for service at concurrent service nodes before joining a queue for the join point. It is possible to have a nested fork-join queue in order to exploit concurrency at as many levels as possible. The resulting concurrency tree is a very appropriate structure to show the concurrency organization in a concurrent programming environment. Also, it is inevitable that there are communications (or synchronizations) between sibling nodes or between parent and child nodes.

Considering such a concurrency tree in a concurrent programming environment, we present a redundancy strategy based on the granularity of any given node in the concurrency tree and the sub-tree that it defines. In this strategy, we duplicate the sub-tree defined by the user over which it is required to implement fault detection. In order to avoid the node explosion that would occur if all nodes in the tee duplicated their descendents, we only allow the master node to create child nodes for both the master and redundant nodes. However, this breaks the synchronization between redundant parent and child, as the redundant child is created by master parent node. This means the redundant parent node will not terminate. To address this, we pair master and redundant nodes at the fork point, and the synchronization between redundant parent and child is achieved via the master parent node.

In scheduling concurrency tree nodes to resources, we ensure that master and redundant nodes are always executed on separate physical resources without any virtualisation. To achieve this, the hardware on which the code is scheduled is defined as fixed pairs of resources, such that although any program may execute on all resources, the master and redundant workloads will always execute on different resources in order to detect both hard and soft errors. For example, if core-1 and core-2 are a fixed pair, then all of the master nodes executing on core-1 will have their corresponding redundant nodes executed on core-2 and vice versa. Hence, although both original and duplicated sub-tree can

be virtualised across all physical hardware resources in the system, for any given node in the tree, the master and redundant copies will always execute on fixed pairs of physical resources.

The System Virtualization Platform (SVP) is a concurrent programming environment designed by the Computer Systems Architecture group in University of Amsterdam. It is a set of system services and language interfaces for the exploitation of concurrency on many-core processor chips. In this concurrency model, each concurrent node in a concurrency tree is called a *thread* and all the same level concurrent nodes that are *created* by a parent node are called a *family* of threads. Every thread can create families of its own, making the model hierarchical. The hardware execution resource is called a *place*, which is allocated at run-time prior to the family being created. Communication and synchronization between threads can occur in two distinct ways. Communication via the single, flat address space in memory is synchronised by the create and sync events, such that data written by a thread prior to create can be consistently read by its descendents and similarly data written by its descendents can be consistently read by the parent following the sync event. In addition to this communication may occur through special hardware-supported channels called *globals* and *shareds*. Global channels are written once by the parent thread, *at any time*, and may be read by any of its descendent threads, i.e. any of the threads in the family created. Shared channels are defined between every consecutive pair of threads in the family. Both types of channel provide data-flow synchronisation.

In this report we have implemented the fault detection framework in the Microgrid execution platform [1]and [25]-[27]. It is a multi-core system that provides dedicated logic able to coordinate single-issue, in-order, multi-threaded RISC cores into computation clusters on chip. It is a highly scalable and configurable many-core architecture. Its machine language provides new instructions to manage concurrency, which are direct implementations of the SVP services. The platform is provided with the SL programming language [28] and [29], which is a interface language to program this platform. SL is designed as an extension to the standard C language (ISO C99/C11). It includes primitive constructs to bulk create threads, bulk synchronize on termination of threads, and communicate using word-sized data-flow channels between threads. It is intended for use as the target language for higher-level parallelizing compilers. Although the fault detection framework is a general one for all concurrent programming environments, we will use some of the dedicated syntax and concepts in the description of fault detection mechanism that come from the Microgrid, SL language, and the assembly language of their cores, since the Microgrid is the experimental platform here.

4.3 On-demand redundancy

We sketch a simple SL function to explain how to implement on-demand redundancy and its related support from the programming model, compiler and ISA. Table 4 shows a simple SL function and how it is augmented with on-demand redundancy support and compiled to a sequence of instructions. This is only part of a complete program. The programming model uses the notation *sl_create* to dynamically define a family of threads on an index range.

SL code:

```
int array[len];
sl_def(sum, void, sl_shparm(int, s))
{
    sl_index(i);
    sl_setp(s, sl_getp(s)+array[i]);
}
sl_enddef

sl_def(t_main, void)
{
   sl_create(,,start,limit,step,
             block,,ftmode,sum,
             sl_sharg(int, s));
    sl_seta(s, 0);
    sl_sync();
    int result = sl_geta(s);
}
sl_enddef
```

Alpha assembly:

<main>:	
allocate	$R_{place}, R_{flag}, R_{fid}$
setlimit	R_{fid}, limit
setblock	R_{fid}, block
create	R_{fid}
puts	0, R_{fid}, 0
sync	R_{fid}, $1
mov	$1, $31
gets	R_{fid}, 0, $1
release	R_{fid}

(a)

<main>:	
allocate	$R_{place}, R_{flag}, R_{fid}$
allocate	**$R_{place'}, R_{flag}, R_{rfid}$**
pair	**R_{fid}, R_{rfid}**
setlimit	R_{fid}, limit
setlimit	**R_{rfid}, limit**
setblock	R_{fid}, block
setblock	**R_{rfid}, block**
create	R_{fid}
create	**R_{rfid}**
puts	0, R_{fid}, 0
puts	**0, R_{rfid}, 0**
sync	R_{fid}, $1
mov	$1, $31
gets	R_{fid}, 0, $1
release	R_{fid}
release	**R_{rfid}**

(b)

<main>:	
allocate/r	**$R_{place}, R_{flag}, R_{fid}, R_{rfid}$**
pair	**R_{fid}, R_{rfid}**
rmtwr	**R_{rfid}**
setlimit	R_{fid}, limit
setblock	R_{fid}, block
create	R_{fid}
puts	0, R_{fid}, 0
sync	R_{fid}, $1
mov	$1, $31
gets	R_{fid}, 0, $1
release	R_{fid}

(c)

Sparc assembly:

<main>:	
allocate	$R_{place}, R_{flag}, R_{fid}$
setlimit	R_{fid}, limit
setblock	R_{fid}, block
create	R_{fid}
puts	0, R_{fid}, 0
sync	R_{fid}, %r1
mov	%r1, %r1
gets	R_{fid}, 0, %r2
release	R_{fid}

(a)

<main>:	
allocate	$R_{place}, R_{flag}, R_{fid}$
allocate	**$R_{place'}, R_{flag}, R_{rfid}$**
pair	**R_{fid}, R_{rfid}**
setlimit	R_{fid}, limit
setlimit	**R_{rfid}, limit**
setblock	R_{fid}, block
setblock	**R_{rfid}, block**
create	R_{fid}
create	**R_{rfid}**
puts	0, R_{fid}, 0
puts	**0, R_{rfid}, 0**
sync	R_{fid}, %r1
mov	%r1, %r1
gets	R_{fid}, 0, %r2
release	R_{fid}
release	**R_{rfid}**

(b)

<main>:	
allocate/r	**$R_{place}, R_{flag}, R_{fid}, R_{rfid}$**
pair	**R_{fid}, R_{rfid}**
rmtwr	**R_{rfid}**
setlimit	R_{fid}, limit
setblock	R_{fid}, block
create	R_{fid}
puts	0, R_{fid}, 0
sync	R_{fid}, %r1
mov	%r1, %r1
gets	R_{fid}, 0, %r2
release	R_{fid}

(c)

Table 4. Shows the compilation of a simple summation function with fault tolerance related parameter `*ftmode* . The three sub-tables show the source code and the assembly for Alpha and Sparc ISAs, For each ISA, the original assembly without fault tolerance attributes (a), the assembly at the redundancy's start point (b), and the regular assembly within the redundancy's scope (c) are shown.

N.b. although this defines the concurrency available, the run-time system (in this case implemented in the core's ISA) is free to run the family of threads sequentially, i.e. in just one or more hardware thread slots. The on-demand redundancy additions to SL and assembly languages when creating the redundant function are highlighted.

Table 4 shows the simple summation function where *sl_create* is augmented to use the parameter *'ftmode*' (i.e. fault tolerance mode). There are two functions: *sum* is responsible for the summation and *t_main* is responsible for creating a family of sum threads to expose the concurrency explicitly in software. The parameter 'ftmode' is the only attribute that is added to the programming model. It is used to determine the redundancy state of the family that will be created. There are three states:

- *NORMAL* - Redundancy is not necessary, so its child family will not be duplicated;
- *START* - Current thread is not duplicated, but its child family will be duplicated. It is the beginning of the redundancy scope in the program;
- *REDUNDANCY* - Current thread is duplicated, and its child family will also be duplicated.

Compiler support is relatively straightforward. It compiles the SL code in listing Table 4 to assembly showed in listing Table 4(a, b, c) according to the fault tolerance mode. For readability, we use symbolic register names rather than numbered registers in the assembly. The listing in Table 4(a) shows the code generated in the normal situation (i.e. without fault tolerance support). The instruction *allocate* attempts to allocate the computing resources (i.e. place, which is a set of cores in the implementation of Microgrid) on which to create the family of threads according to its parameters: place identifier and some other flags in R_{place} and R_{flag}. If successful, the family identifier *fid* of the allocated family entry on the first core will be written into R_{fid}. Then the properties of this family will be set through *set* instructions, such as the total number of threads, and the limit on number of hardware threads per core. Finally, this family of threads will be created using the instruction *create,* see [1] for more details.

The listing in Table 4 (b) presents the assembly generated at the redundancy scope's beginning. Generally, the compiler will duplicate all the instructions relating to family creation and initialization, such as *allocate*, *set*, *create*, *put*, etc. However, there are some issues that need to be considered. The instruction *allocate* for the redundant family should generate a different place identifier compared to the master family, as we dispatch each to a different core for hard error detection. The master and redundant families need to be connected though the instruction *pair*. The most important point is that the current thread, which is executing the assembly of the listing in Table 4 (b), only synchronizes with its master child family. Also the data returned from the child family of threads read by the instruction *gets* will not be verified against its redundant child family therefore the thread reads from its master child family only. This is because the redundancy scope began from the current thread's child and hence the current thread is out of redundancy sphere and is not to be protected.

Table 4 (c) gives the assembly created within the redundancy scope. Because it is within the scope, it means that both redundant and master threads will execute it. So we do not extend the assembly as for the redundancy start point, which is executed by one thread only. Three instructions, which are italic bold in the code, are extensions to the ISA that give an elegant thread duplication solution in our concurrent programming environment. More details about the thread duplication strategy within redundancy scope and these extended instructions will be discussed in Section 4.4.

The fault tolerance mode can only transit from *NORMAL* to *START*, to *REDUNDANCY*. *START* is a temporary state that occurs once in the transition from *NORMAL* to *REDUNDANCY*. The transition is irreversible, as it is not worth supporting a reversal mode transition in hardware. Note that the reverse transition occurs naturally on termination of the family created in fault tolerant mode. In effect, what this scheme does is to label a node in the concurrency tree and every thread below this label is within the sphere of redundancy. This can be done at any number of nodes to create redundancy for those critical regions. Thus the correct redundancy scope can be easily achieved with much lower design complexity at the software level by arranging thread structure properly.

4.4 Intelligent redundant thread creation and synchronization

As described in section 4.3, the thread duplication strategy cannot be propagated without change from the start point to the body of the redundancy scope because only one thread executes the thread duplication code at the start point, but in the body of redundancy, both master and redundant threads execute the thread duplication code. Obviously, we cannot allow both master and redundant threads to create their own child family, as this will lead to an explosion in the number of threads created, whereas what is required is a single copy of each thread in the concurrency tree.

As a result, allocation of new resources is not allowed in redundant threads. The master thread will allocate double the resources as before but the redundant thread will do nothing (i.e. allocate in Figure 20). However, usually a redundant parent thread has to wait for the synchronization of its children to continue or end its execution (i.e. sync in Figure 20). We use the master parent thread to bridge this gap. There are two family identifiers returned for every allocation in the master parent thread. One will be sent to the redundant parent thread so the redundant parent thread eventually gets its resource from the corresponding master parent thread without doing any allocation. This is achieved through the communication protocol shown in Figure 20. The synchronization link is built between redundant parent and child family. This is the main idea behind the intelligent, redundant thread creation and synchronization. Because of this, all the other instructions related to thread creation run without any modification.

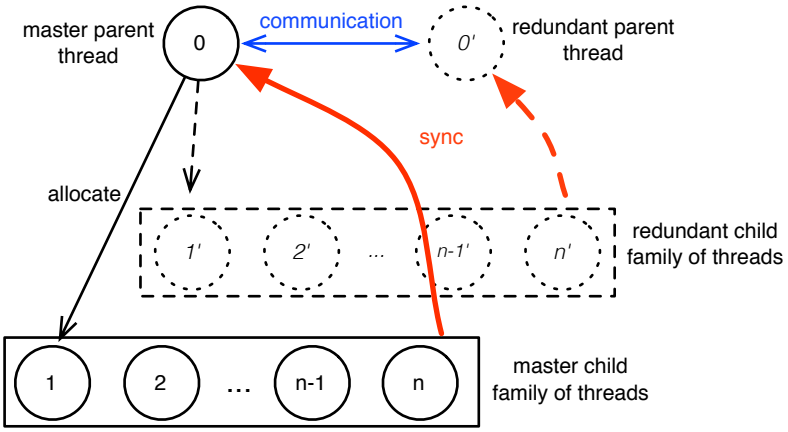


Figure 20. Intelligent redundant thread creation and sychronization

This strategy is supported in the ISA and at the micro-architecture level. Three new instructions are added for thread duplication and its communication, over and above those used already to support the SVP concurrency management in hardware. However, these new instructions have a different operation in master and redundant threads, as shown in Table 5. The instruction *allocate/r* in the master thread is responsible for allocating cores for both the master and redundant child families. The redundant thread only waits on its register for the redundant child family's identifier, which is sent by the master thread. Then the instruction *pair* in the master thread makes the master and redundant child family known to each other, which is necessary for the communication between master and redundant threads shown in Figure 20. Finally, the redundant child family's identifier is returned to the master parent thread and will be sent to redundant parent by the instruction *rmtwr*. There is no operation for *pair* and *rmtwr* in the redundant thread. Up to now, two places for the master and redundant child families are allocated, and the master and redundant parents receive master and redundant child family identifier, respectively. Considering the assembly in listing Table 4(c), all the subsequent thread creation related instructions are depend on its family identifier, R_{fid}. So all these instructions do not need to be modified to fit thread duplication, as we have already connected parent thread and child family in both the master and redundant group, although both master and redundant child family are allocated by the master parent thread.

Instruction	Master thread	Redundant thread
allocate/r	Send an allocation message to the place with flag. The master family identifier will be returned to R_{fid}, the redundant family identifier to R_{rfid}.	Set the output register R_{fid} to pending; Send R_{fid}'s index to master thread*.
pair	Send a message to the destination place, which will pair master and redundant families.	No-op
rmtwr	Write the redundant family identifier to the redundant threads R_{fid}, which was pended by instruction *allocate/r* in redundant thread.	No-op

* The creation process of master and redundant threads is lockstep to ensure redundant thread receives master thread's identifier. The executions of these two threads are independent.

Table 5. new instructions and their operation

4.5 Thread Pairing

In order to support redundant thread creation and synchronization, the master and redundant threads should be paired. As threads are organized in units of families, master and redundant families should be paired as well. We add the instruction *pair* to pair master and redundant families in an explicit way. In contrast, thread pairing is implicit and implemented in micro-architecture level using a quasi-lockstep method.

This thread pairing technique supports dynamic allocation of resources in the constraint of real-time run-time environment. It can guarantee both master and redundant families are executed in the same model, either sequential or parallel. Both master and redundant families execute parallel code when their resource requests are satisfied, or they will execute sequential code instead. Also, the thread pairing technique allows master and redundant families have different size of hardware multithreading window.

4.5.1 Family Pairing

We use instruction *pair* to implement the family pairing explicitly. The format of instruction pair is:

```
pair Rfid , Rfid'
```

The instruction *pair* checks the values of R_{fid} and $R_{fid'}$ and if they are not both valid, it will set both of master and redundant family identifier as invalid to indicate that resource allocation has failed. Consequently, both master and redundant families will execute sequential code (as in normal execution), as there are not enough resource for their parallel execution. If both R_{fid} and $R_{fid'}$ are valid, master and redundant families will execute the parallel code and the instruction pair will send two similar pair messages to master and redundant families respectively. When a family receives the pair message, it will set its matching family identifier in its family entry, and transfer this message to its next family on another core.

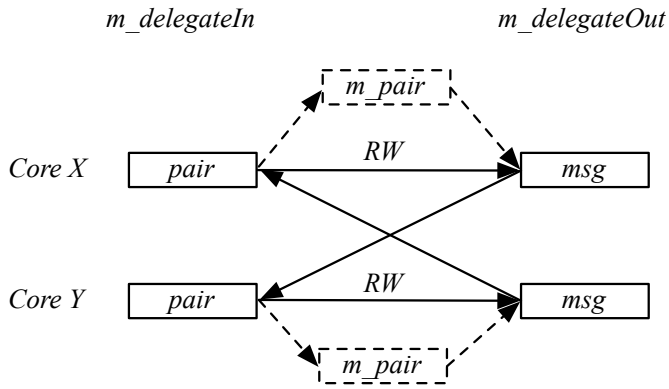


Figure 21. Deadlock in the delegation network and its solution

In this way we guarantee that both master and redundant families execute the same code, which depends on the value of family identifier. So the family identifier should be reset in the instruction pair as mentioned above. In other words, the registers R_{fid} and $R_{fid'}$ are not only input but also output registers in the instruction. This means that the pair message will generate a remote register write message in the last core of the family.

In the delegation network of the Microgrid, there is one entry for each input and output of a core. The pair message may cause deadlock as shown in the solid line shown in Figure 21. In the cycle t, *msgs* in *m_delegatOut* of both core X and Y are going to be written into *m_delegateIn* of the other core. But both pair messages produce new remote register write messages, which are going to be written to the corresponding *m_delegatOut*. Hence we have deadlock.

In order to prevent deadlock, a buffer *m_pair* is added in each core to buffer pair message. In Figure 21, the dotted line replaces the solid line tagged as RW. The size of *m_pair* is same with the size of family table, which means this buffer can keep all pair messages even if family table is fully used. Theoretically, the mutual exclusion condition is removed as we provide one entry of buffer to each pair message. There is only one pair message in a family.

4.5.2 Thread Pairing

Thread pairing is more complex than family pairing as it is dynamic and occurs during thread creation and scheduling. We implement the thread paring implicitly during their creation.

The lifetime of thread is split to four stages as shown in Figure 22: *allocation*, *activation*, *execution* and *termination*. Thread creation includes allocation and activation. In the allocation stage, a thread entry, thread local storage, and registers are allocated for a thread. When the allocation has succeeded, this thread will be queued in the ready list. Here, instructions of threads in ready list are requested and the thread will be pushed onto active list when the instructions are loaded in the L1 I-cache.

We use a quasi-lockstep method to pair threads at creation as shown in Figure 22. At the allocation stage, a redundant thread can be allocated as usual. When

the allocation of the redundant thread is successful, a message (see communication I in Figure 22) containing the master family identifier is sent to the master family that will increase the redundant thread creation counter in master family. The same allocation in the master thread can only proceed with the condition that the redundant thread creation counter is lager than 0, notifying that the redundant thread was already allocated in the paired core. It is because of this that both master and redundant families have a sequential thread creation procedure. It decreases the redundant thread creation counter after the master thread allocation has succeeded.

After allocation, the master thread will send a message (see communication II in Figure 22) to the paired core, which contains the master thread identifier and logic index. The paired core searches the thread ready list looking for the redundant thread, which is related to this master thread and sets the master thread identifier field in the thread entry of the redundant thread. Up to now, the thread pairing is done, and it is assymmetric. Only the redundant thread knows its master thread, but the master thread does not know its redundant thread.

At the activation stage, the redundant thread will check the master thread identifier (*mtid*) parameter in the thread entry first. The nonempty *mtid* ensures the instruction *allocate/r* in the redundant thread can send its R_{fid}'s absolute address to the master thread. If the redundant thread is scheduled in the ready list with an empty *mtid* field, it will be pushed into a specific ready list, which is created only for buffering redundant threads without *mtid*. This prevents deadlock caused by redundant thread when it stuck at the ready list due to *mtid* being invalid.

Master and redundant threads are enforced to use same thread local storage (TLS), which can decrease the fault detection latency. So we specify that only the master thread can reserve and un-reserve TLS. The reservation and release of TLS occurs at the allocation and termination stage of the thread respectively. As we have a quasi-lockstep creation between master and redundant threads, the master thread can reserve TLS before the execution of both master and redundant threads. In order to make sure that the master thread releases TLS after the termination of the redundant thread, the termination of the master thread should only proceed after it receives the termination message from the redundant thread, see communication III in Figure 22. The execution of the master and redundant threads are independent.

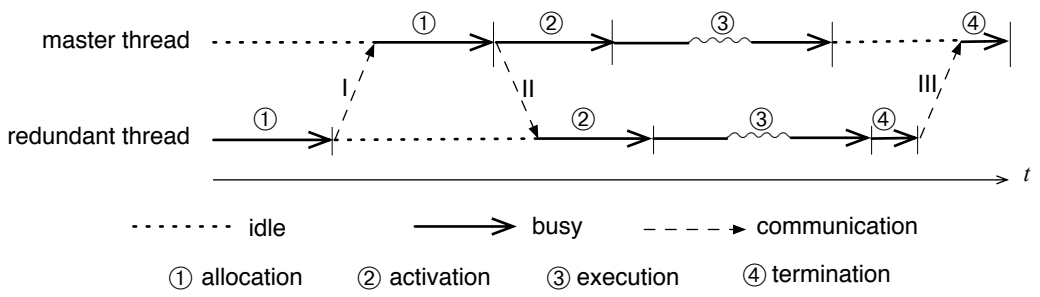


Figure 22. The lifetime of master and redundant threads

The resource allocation strategy in Microgird is dynamic and depends on the current run-time environment. This means that the master and redundant families may have different resources, such as hardware multithreading slots, as they run on different cores independently. Under this quasi-lockstep creation, master and redundant family can be executed correctly even if they have different sizes of their hardware multithreading window. In other words, this thread pairing technique allows master and redundant families to have different maximum number of concurrent threads (i.e. block size).

4.6 Output comparison

Like the other thread-level redundancy techniques, we must also compare the results of the master and redundant threads to detect faults. To achieve this, a comparison buffer is added between the L1 D cache and secondary memory. This buffer is shared by a core pair. As stated above, the core and private L1 cache are contained in the sphere of replication in our fault detection framework. This means that the other components by definition are out of the sphere, such as L2 cache and off-chip memory. These are assumed to be fault free. Each output (i.e. *store*) should be stored to both L1 and comparison buffer first, then compared in comparison buffer before being committed to secondary memory. The operation of data input (i.e. *load*}) is the same as before: the data come from secondary memory to L1 D cache. In this report, we do not address the issue of supporting output comparison in I/O. However, we note that if this is memory mapped, then the same mechanism for comparison that supports stores will be available.

As the redundant thread knows its master thread's identifier, which is explained in Section 4.4, the comparison buffer is organized as a number of sets that are indexed by both its core and master thread identifier. Furthermore, the master thread writes data to the set specified by its identifier in the comparison buffer, which is owned by the core it runs on. A store in the redundant thread writes data to the set specified by its master thread. For example, if the master thread (thread identifier is t_i) runs on core i, and redundant thread (thread identifier is t_j) runs on core j, then the stores of these two threads will be sent to set t_i in the comparison buffer specified by core i.

Each set is a FIFO queue, as thread instructions are executed in order in the Microgrid platform. This means that all the stores in one thread will be appended to its dedicated set and compared in order. Each entry in a set has three fields:

1. The address of store;
2. The value of store;
3. The status flag (2 bits):
   - 00 -- empty line;
   - 01 -- written by master thread;
   - 10 -- written by redundant thread.

When a set of comparison buffer receives data, the data will be written to the set directly if the set is empty. Otherwise, it will check whether the data and the head of the set come from the same thread. If they are, the data will be appended to the end of the set. If they come from different threads, which mean master and

redundant thread, then they will be compared. A fault is detected when they do not match. If they do match, which shows the results are correct, the data will be popped from the set and written to secondary memory. The pseudo code for the output comparison is shown below.

```
function comparison (data){
if set is empty
   append data as head of queue;
else
   if set->head.flag is NOT equals data's flag
      compare set->head with data;
      if they match
         write data to secondary memory;
         pop the head of queue;
      else
         a fault is detected;
   else
      if set is full
         suspend current thread;
      else
         append data to the end of queue;
}
```

Any read request coming from L1 cache, will first search the set indexed by the current thread. Data will be returned if it is available, otherwise the read request will be sent to secondary memory as usual. The comparison buffer does not change the memory protocol, which means it can be used with the Microgrid's various memory interconnects.

4.7 Experimental results

4.7.1 Experimental platform

The thread-level fault detection framework is implemented in the Microgrid. Figure 23 illustrates a Microgrid chip with a configuration of 128 cores, which is the base platform we have used for all experiments. All cores on chip are organized in a linear partitionable network for resource allocation and concurrency management within an allocated cluster. It is worth mentioning that there is a custom distributed cache protocol derived from [31][32]: memory stores are effected at local L2 cache and updates are propagated and merged with other copies. Upon an explicit barriers or bulk creation or synchronization of threads, the update acknowledges must be counted by thread or family to ensure memory consistency for the programming model. The hardware parameters most relevant to this report are shown in Table 6, more details can be found in [25][26][27].

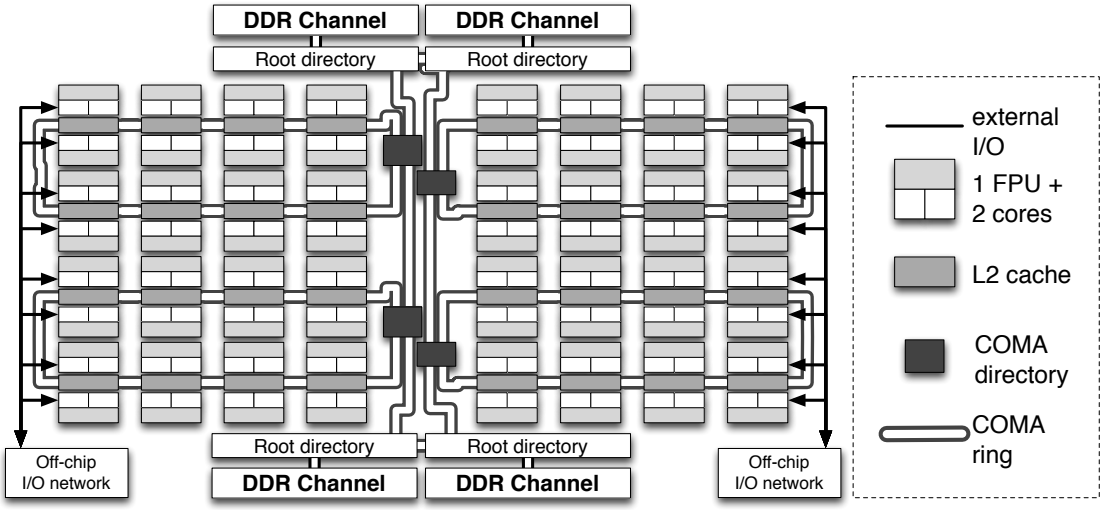


Figure 23. The Microgrid chip with 128 cores

Components	Specification
Core	• Alpha ISA with SVP extensions • In-order pipeline of 6 stages • 1024 integer registers and 512 float registers • 1.0 GHz frequency
L1 D Cache	• 16 sets with 4-way set associative • 4KB capacity • Write through
L2 Cache	• 512 sets with 4-way set associative • 128KB capacity • Write update • Shared by 4 cores via a snooping bus
Distributed Cache structure	• Each sub ring has a directory and 8 L2 caches • 4 root directories each connects to a DDR3-1600 channel mapped to a DRAM bank • Ring directories and evenly distributed root directories from the top ring

Table 6. The specification of Microgrid

The simulator currently executes benchmarks in which the redundancy scope is the complete benchmark, however, we only execute small kernels not large applications. Selective redundancy and fault coverage are left to future work. The six benchmarks include image processing kernels, FFT and encryption and are used to evaluate the thread-level fault detection technique shown in Table 7.

Category	Benchmarks	Instructions count	Description
Image process-ing	convolution	28 million	The size of original image is 800*400. It is zoomed in to 6400*3200, and reduces each pixel from 24 bits to 8 bits in grey conversion.
	zoom in	29 million	
	grey conversion	11 million	
Signal process-ing	FFT	19 million	Use a 64K phase lookup table and butterfly reduction.
Encryption	rc4	0.05 million per stream per core	The problem size is scaled with number of cores and hardware threads per core, i.e. 1Kbyte stream per hardware thread.
	seal	0.5 million per stream per core	

Table 7. Description of benchmarks

4.7.2 Results

We use the performance of non-redundant benchmarks run in Microgrid as the baseline and we call the non-redundant benchmark the base benchmark, and benchmark with complete redundancy scope the redundant benchmark. We evaluate the performance overhead when redundancy is introduced. The performance overhead of a redundant benchmark is also called performance penalty in this paper, which is defined as follows:

$$performance\_penalty = \frac{t_{redundant}}{t_{base}} - 1$$

And $t_{redundant}$ is the execution time of redundant benchmark, t_{base} is the execution time of base benchmark.

The master and redundant threads are always distributed to different cores that are fix-paired. We should distinguish the experimental results of the base benchmark run on single core from that running on many cores. For the single core base benchmarks, another core is used for the redundant execution, which means a redundant hardware resource is added. However, there is no redundant hardware resource added for many-core base benchmarks. For example, if the base benchmark has 2 independent threads in total, and thread 0 and 1 run on core 0 and 1, respectively. Then its redundant benchmark still runs on core 0 and 1 with master thread 0 and redundant thread 1 run on core 0, and master thread 1 and redundant thread 0 run on core 1. It shows that more resource contention occurs in many-core redundant benchmarks.

Single core. In Figure 24, the bars from left to right for each benchmark correspond to block size or number of streams (i.e. the number of threads per core). This ranges over 1, 2, 4, 8 and 16, all later figures are organized like this. Figure 24 shows that the performance penalty is less than 11\% for all scenarios, which is mainly caused by output comparison. As the number of threads per core

increases, the performance penalty reduces because the latency of the output comparison can be tolerated by the hardware multithreading mechanism. It can be seen that in some cases the performance of the redundant benchmark is better than that of the base benchmark. We believe this is because the comparison buffer moderates peak traffic rates onto the memory network. We have, in past experiments, observed a degradation of performance at large numbers of threads on these benchmarks due to correlated stores in many threads saturating the memory network and causing a higher latency on synchronization.

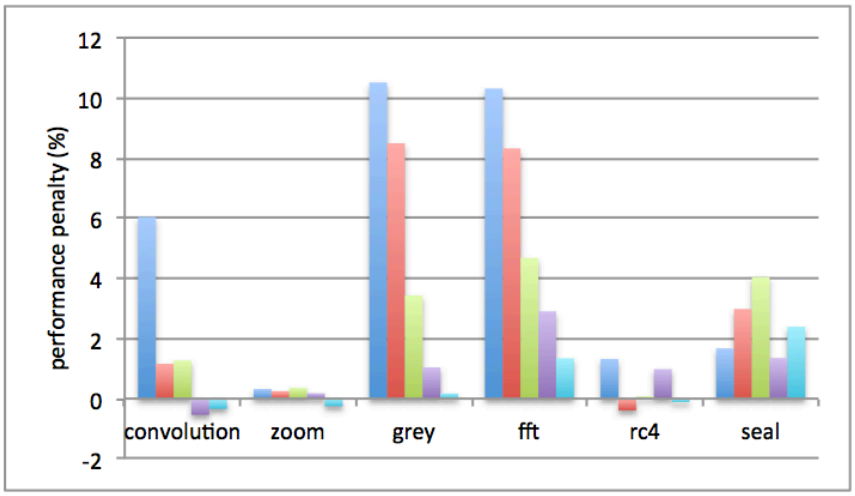


Figure 24. Performance penalty of single core base benchmarks

Many cores Before discussing the result of benchmarks run on many cores, we would like to give a definition and a proposition, which is related to the results analysis.

Definition. The parallel efficiency of a program is the ratio of execution time of the concurrent part to the whole program.

For example, the concurrent and sequential parts of a program have an execution time of 0.4 and 0.6 respectively, when executed sequentially. The execution time of sequential part is constant but the execution time of the concurrent part becomes 0.2 or 0.1 when 2 or 4 cores are used. Therefore the parallel efficiency is 0.4, 0.25, 0.14 for 1, 2 and 4 cores.

Proposition. The performance penalty of a redundant benchmark running on many cores in the Microgrid is approximately equal to the parallel efficiency of its base benchmark.

For example, if the sequential execution time of the concurrent and sequential part in the base benchmark is X and Y, considering the situation where the base benchmark is run on N (N>1) cores, the parallel efficiency in the base benchmark is approximated by:

$$efficiency_{base} \approx \frac{\frac{X}{N}}{\frac{X}{N}+Y}$$

And the execution time of base and redundant benchmarks may be approximated by:

$$t_{base} \approx \frac{X}{N}+Y$$

$$t_{redundant} \approx \frac{2X}{N}+Y$$

The equations above are approximation as contention, scheduling and latency tolerance among concurrent parts needs to be considered. As we know, the redundant benchmark has twice the amount of work, so its execution time is approximately twice that of the base benchmark (i.e. $\frac{2X}{N}$). However, the sequential part is always run on single core, because of the allocation strategy of Microgrid, and the cost of single core redundancy is small as mentioned before. So we still use Y as an approximate value. Finally, we can get the performance penalty of a many-core redundant benchmark as follows:

$$performance\_penalty = \frac{t_{redundant}}{t_{base}} - 1.$$

The performance penalty varies in the many-core benchmarks (Figure 25) and is related to the original efficiency of the base benchmarks and the distributed cache system. Compared to the base benchmark, the performance penalty of the redundant benchmark is always less than 100% even though it executes double the number of instructions using the same single-issue, in-order core resource. This is mainly attributable to the hardware multithreading and its ability to tolerate latency.

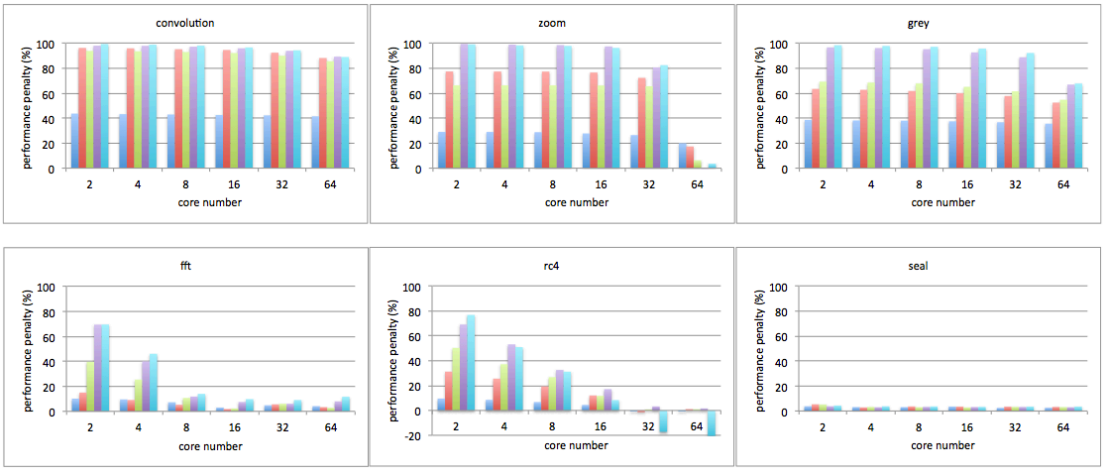


Figure 25. Performance penalty of many cores base benchmarks}

The block size in Figure 25 is based on the base benchmark. For example, a scenario of N cores with a block size of M indicates that the base benchmark is executed on M threads in each of N cores. However, the redundant benchmark is executed in N cores with 2*M threads. The additional M threads are used by the redundant copy. The performance penalty of 1 thread per core is always smaller than other situations. It is less than 44% even in the computationally intensive *convolution* program. It also relatively stable for different numbers of cores. Going from 1 to 2 threads per core gives a significant speedup and it is shown that a speedup of between 1.5 to 2 can be achieved for all the benchmarks here.

The general trend we can find in Figure 25 is that the more cores and threads that are used, the less the performance penalty. Because for a given program, the parallel efficiency becomes lower when more resources are used. The results therefore confirm the prediction made above.

The relation between block size and performance penalty is not so clear, as the distributed cache system is involved. Usually, the larger the block size, the greater the performance penalty, because the efficiency of the base benchmark is higher when a bigger block size is set. We also see some unexplained results with a block size of 16 in *zoom* and *rc4*. We believe the reason is again as described above in the case of the single core benchmark. The more cores used the greater bandwidth demanded of the memory system, which makes it more likely that there will be an impact from correlated stores on synchronization costs. It appears that the comparison buffer mitigates this problem. Note that in any case the redundant benchmark has twice the computational complexity (operations per byte stored) as the base benchmark but only one store to the L2 cache occurs for the two writes, one from each core.

We took *rc4* as an example using 64 cores and ran some diagnostics to try to explain this anomaly in more detail. We found that the messages in the cache ring decreased by 55% (read-related by 75% and write-related by 44\%) in the redundant benchmark compared to the baseline, even though the total number of read requests has doubled in the redundant benchmark (n.b. the number of write requests is same). We believe that because the comparison buffer delays each write it will relieve the congestion of cache ring. Remember that the cache policy on L1 is non-allocating write through with a write update policy at L2. So the comparison buffer acts like a write combining buffer delaying and aggregating writes. In effect it allows more writes to be achieved without generating an update message in the ring. Similarly as the combining buffer delays the eviction of the cache line it can also increase the hit rate to existing lines. Note that the encryption benchmarks have little locality between threads, as each is an independent stream. Thus the redundant benchmark is able to improve both the write and read hit rate and this is why the performance penalty goes lower with the increasing of block/stream size in 64 cores of *zoom* and *rc4*. Moreover, the *rc4* redundant benchmark can have better performance than baseline even though it has double number of threads and twice the work to do. It is in effect highlighting the inefficiency of the distributed cache system, which is sensitive to the quantity and distribution of messages at high load.

The three image processing redundant benchmarks have a higher performance penalty compared to the other benchmarks, as they are highly parallel benchmarks with exceptional locality and hence have high parallel efficiency. Although FFT is also high parallel, its communication is non-local, which means that its parallel efficiency is constrained by contention in the cache ring. So again, its redundant benchmark has much smaller penalty than image processing ones. The encryption benchmarks have sufficient parallelism (one stream per hardware thread) but lack locality as each stream is independent. Thus frequent evictions of data cause the parallel efficiency of the base benchmark to be low and the overhead of the redundant benchmark to be correspondingly low.

Finally, we note that all of the results confirm our proposition that the redundant benchmarks' performance penalty is strongly related to the parallel efficiency of base benchmarks.

4.8 Proposal for Fault recovery

4.8.1 Background discussion

In the above sections we have outlined a method in the Microgrid to identify faults by executing paired threads on different processors automatically, when a fault-tolerance option has been set by the programmer for a particular sub-tree of the computation. However, when a fault is identified, it is necessary to initiate a recovery from that fault as seamlessly as possible.

For fault recovery, it is necessary to capture all state from the unit of program being recovered from and to commit this to outside only when it has been confirmed that the unit being protected has completed successfully. Fault recovery can be implemented in software or in hardware, or indeed a combination of the two. Obviously re-execution of large units of code because of an error has a time/energy impact. Hardware recovery may allow a finer grained recovery and hence minimise this impact by constraining the re-execution to smaller units of code. However, the later needs a hardware buffer for check-pointing that comprises all processor state and memory side-effects even if the latter are intermediate results. Also it loses performance due to checkpoint saving and synchronization when the execution is fault-free. Furthermore, its hardware overhead is heavy when the checkpoint content is large and complex. In contrast, software recovery is coarser grained and does not need dedicated hardware for checkpoint logging. It has no hardware overhead and hence simplifies the hardware design. However, it may have long recovery time as it may re-execute non-faulty parts of the program due to its coarse grained nature.

Another issue in the Microgrid is that because of its concurrency management strategy it is extremely complex to implement a hardware recovery strategy for as the concurrency management unit in each core produces too much data to checkpoint. Meanwhile, we want to avoid a substantial cost to performance in software recovery. However, we believe this is possible to implement some hardware support for recovery using only the buffering we require for fault detection, the constraints are on the complexity of the threads being recovered from. Many Microgrid applications have few memory side effects per thread at

the leaves of the concurrency tree and some may even tolerate errors intrinsically.

Therefore we propose a fault recovery strategy that has a low impact on performance combined with minimal additional hardware. This proposal represents a co-design of hardware and software to implement a complete fault recovery strategy in code protected the fault-tolerant scheme identified above. It restores a faulty program using different schemes according to the memory side-effects and dependencies of a program. This scheme combines a partial thread-level hardware recovery with user-defined software recovery. The partial thread-level hardware recovery re-executes a faulty thread to recover from a detected fault in hardware. It does this automatically when the thread is independent and has few side effects. It tries to make common situations fast but provides only partial coverage. It is then the responsibility of the software to provide check-pointing and recovery when hardware recovery fails or when an error occurs that cannot be recovered from, i.e. where the thread is too complex for partial thread-level, hardware recovery.

A dedicated software approach to recovery is likely to be considerable more efficient than the same approach adopted in hardware for the following two reasons:

1. Not all memory side effects need to be buffered, some will be temporary results of a computation; hardware has no knowledge of this and must capture all side effects before committing them, a software recovery scheme will be aware of which data is live and which is not.

2. Often with iterative or many stage algorithms, new data overwrites old and hence must be buffered; in a software recovery using an alternating, two-buffer scheme, no copying overhead for the commit stage is necessary. For example instead of A->A->A->A in an iteration, a software recovery scheme can compute: A->B->A->B where A and B are arrays of data. Additional storage is required but if any stage fails it is simply re-executed from the original input data without any copying on commit.

The main contribution of this proposal is that it maximises the benefit from both hardware and software recovery while avoiding their respective drawbacks. We use hardware recovery for fine-grained recovery with less performance cost recovery when the checkpoint requirements are small. At the same time, we use software recovery to back up and provide a full recovery scope in case that any errors in the program are out of the scope of the hardware recovery scheme due to the complexity of their checkpoint requirements.

4.8.2 Fault-recovery proposal

The checkpoint buffer is the main constraint in hardware recovery. And memory side effects are the main part of a checkpoint. All memory writes should be committed only after checkpoint validation. Many applications that continue to drive increases in chip performance include computer vision, data mining, search, media processing, and data-intensive scientific applications. Many of these applications can be expressed concurrently for the Microgrid, using many independent threads and where each thread has few memory side effects. That

makes it easy to recover from a detected fault in a thread using automatic re-execution of the offending thread in hardware, without checkpoints, as memory transactions for each thread are buffered in the fault-detection comparison buffer.

Our proposed partial thread-level re-execution recovery scheme in hardware automatically targets these emerging applications. For more complex applications, where thread may specify more writes to memory or when threads are dependent upon each other and make use of thread-to-thread communication, the applications or units of them will be protected by software recovery. Software recovery then, is the last line of defense when hardware recovery fails. This is summarised in Figure 26.

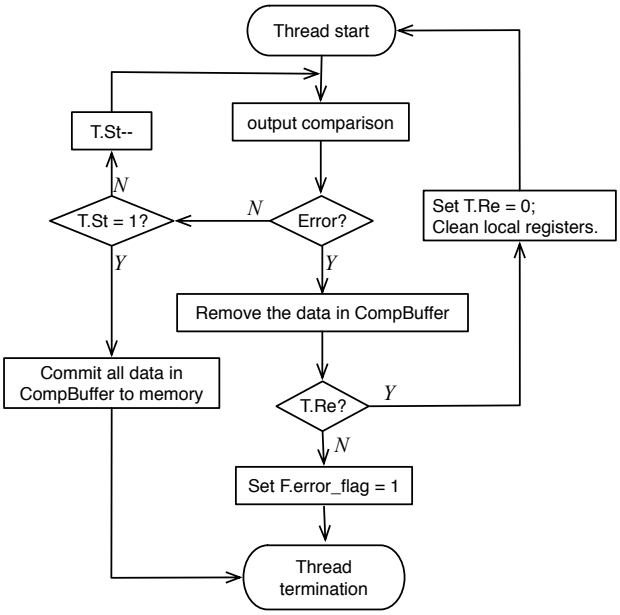


Figure 26. Overview of the hardware/software co-design approach to fault recovery in Microgrid

4.8.2.1 Partial thread-level hardware recovery

Partial thread-level hardware recovery is a scheme where thread will be re-executed automatically for recovery without software intervention when a fault is detected. Only a re-executable thread can be recovered by the partial thread-level hardware recovery scheme. A re-executable thread is a thread that satisfies all of the following conditions:

1. the thread is independent, i.e. does no communicate to other threads by register sharing or copying;
2. there are no branches within the thread, otherwise the number of stores made can not be bounded;
3. there are no family creation within the thread;
4. global registers must not changed during thread execution, note that this is a requirement for determinism as any thread in a family may update a global and in any order;
5. the thread has few store instructions, where the number may be limited by implementation.

Programs such as: matrix multiplication, image processing, FFT etc, all comprise mostly re-executable threads.

The above conditions are checked on compilation. A thread will be tagged as re-executable when all five conditions are meet. The compiler also counts the number of store instructions, which is used in the output comparison stage. In practice, all stores from a thread are kept in the comparison buffer until all writes from the paired threads have been verified.

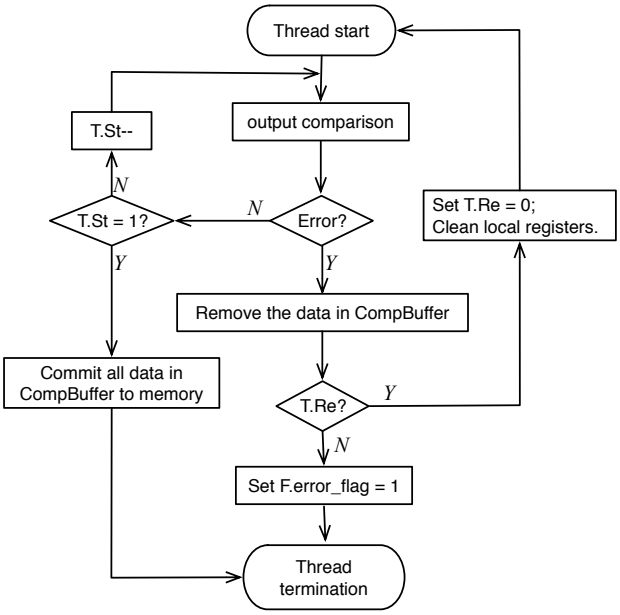


Figure 27. Partial thread-level hardware recovery process

From implementation perspective, we need to set a threshold (e.g. 4) for number of store instructions, which are checked, i.e. condition 5, in compilation which is constrained by the implementation choices. Also, two parameters are added to all thread contexts:

- *Re* (1bit): indicating whether the current thread is re-executable or not.
- *St* (2bits): a counter of the number of store instructions in the thread.

The partial thread-level hardware recovery process is shown in Figure 27. Once an error is detected, all the data in the comparison buffer for the current thread is removed. Then the thread will be restarted if it is a re-executable thread. Before thread re-execution, the local registers of the thread are cleaned as they may be polluted by the last faulty execution. Meanwhile, the re-executable flag of the thread is reset as thread re-execution is only allowed once. If the recovery fails or the thread is not re-executable, it will set the error flag in family table entry, which can be monitored in software.

4.8.2.2 Software recovery

For programs outside of the scope of partial thread-level, hardware recovery, we support a software recovery scheme. In general, we expose the error from hardware to software and make the software responsible for recovery. We use

family synchronization to expose the hardware fault to software if there is a fault.

When an error is detected in a thread, it will check whether current thread is re-executable or not. If it is, the thread will be re-executed in hardware automatically. If it is not a re-executable thread or if the re-execution recovery fails, then the thread will expose this error to family by setting the error flag in the family entry. In addition to this the hardware must pre-empt any continuing computation as we can safely assume any data produced will be discarded in the software recovery. As pre-emption is implemented in hardware, no signal need be raised or acted upon. The normal synchronisation event will occur sooner and the error code can be checked by the code at sync to determine any subsequent action. Currently the Microgrid supports two forms of pre-emption:

1. Break a family. This soft pre-emption causes the thread creation process to cease mapping software threads to hardware resources and then allows all currently executing threads to terminate normally. This is not implemented recursively and if a running thread creates new threads, these must all be executed before the thread terminates. It is useful for families of leaf threads such as the stages in an FFT.
2. Kill a place. Currently not implemented but is the better option so long as the unit of software being protected has the exclusive use of a place (a set of cores).

We propose to make the choice of pre-emption software selectable, so that when an error occurs the programmer has already selected which of the above options will be used for pre-emption. When a family synchronizes, the error flag is checked. If the error flag is set, the family will transfer the error information to software through the synchronization. Finally, the programmer is responsible for check pointing and software recovery.

A program with software recovery support is shown below:

```
sl_create (, , start, limit, step, block, sl__ft, sl_kill, func);

sl_sync(error_flag);

if (error_flag)

      then recovery;
```

The keyword parameter *sl__ft* shows that the family will be created is in the scope of redundancy, i.e. the family is protected by fault detection and recovery. The flag *sl_kill* indicates that the programmer wishes to kill the place rather than the default of using break. The error_flag is transferred from to hardware to program, and is used as a condition by the software to initiate whatever recovery process is necessary. Note that this code is simplified as other information can also be passed back via the sync code.

4.9 Conclusions on Thread-level Redundancy

Fault tolerance will be inevitable as multi-core systems become mainstream. There is no doubt that concurrent programming is also an important opportunity to improve the efficiency of multi-core processors. In this context, this paper

presents and implements an on-demand fault detection framework that has been added to an existing multi-threaded, many-core chip emulation.

By necessity the framework has been co-designed across multiple layers, including programming model, compiler, ISA and micro-architecture. This cross-layer cooperation makes the fault detection and recovery mechanism much more flexible and efficient. It also makes on-demand redundancy possible. When required there is automatic creation and synchronisation of both master and redundant child families. The scheme requires only a flag in the instruction creating the family and the compiler and hardware cooperate to create and link these threads created. It also avoids any redundant thread creation by allowing only the master threads to continue to create further threads and in doing keeps the synchronization channel between redundant parent and its child threads.

Experiments presented in evaluating this scheme show results on the overhead of executing the redundant thread across a range of kernel benchmarks with a range of concurrency resources (threads per core and numbers of cores). We have presented and verified a proposition that the overhead of redundant execution is strongly dependent on the parallel efficiency of the base benchmark. The results show that for a single core benchmark, where additional resources are brought into play, the performance penalty for redundancy is less than 11%. For the many-core situation, the penalty is never larger than 100% even though the redundant benchmark has twice the dynamic instruction count. This is attributed to the latency tolerance of hardware multithreading, especially in situations where the parallel efficiency is poor on the baseline. In some cases we even show that the redundant benchmark has a better performance than the baseline and confirm that it places a smaller load on the shared resources due to a modified scheduling of writes to L2 cache.

Although not implemented in the emulation platform, the report also outlines a strategy for fault recovery. This uses a combination of hardware and software support, the latter provided by the programmer of the application. The hardware support does not require any additional hardware but relies on the master/redundant thread comparison buffer used in this scheme. It delays writes to L2 cache from simple threads with few writes and is able to restart the thread if an error is detected. Because of the nature of the programming model for the Microgrid, where all independent computations are exposed as threads (e.g. all independent loop iterations), this covers the common case. For threads that are not "simple", when an unrecoverable error is detected the remaining work in the region being protected is aborted a return code is passed to the code through the synchronisation primitive, this allows the programmer to write code in a way that allows efficient check pointing on these regions of code.

This work will be continued by Fu Jian at the University of Amsterdam and further results on the implementation of a complete fault detection/recovery and simulation using fault injection may be of future interest to ESA.

5 References

[1] Mike Lankamp, Raphael Poss, Qiang Yang, Jian Fu, Irfan Uddin and Chris R. Jesshope (2013) *MGSim - Simulation tools for multi-core processor*, University of Amsterdam, **arXiv:1302.1390v1 [cs.AR]**, http://arxiv.org/abs/1302.1390

[2] J. M. Calandrino and J. H. Anderson (2008) Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study, in *Proc. 20th Euromicro Conf. on Real-Time Systems*, ECRTS '08, pp. 299-308.

[3] H. Leontyev and J. H. Anderson (2007) Tardiness Bounds for EDF Scheduling on Multi-Speed Multicore Platforms, in *Proc. 13th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications,* RTCSA 2007, pp. 103-110.

[4] J. H. Anderson, J. M. Calandrino and U. C. Devi (2006) Real-Time Scheduling on Multicore Platforms, in *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS 2006, pp. 179-190.

[5] J. M. Calandrino, D. Baumberger, L. Tong, S. Hahn and J.H. Anderson (2007) Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms, in *Proc. 13th IEEE Real Time and Embedded Technology and Applications Symposium*, RTAS '07, pp. 101-112

[6] S. Euiseong, J. Jinkyu, P. Seonyeong and L. Joonwon (2008) Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors, *IEEE Transactions on Parallel and Distributed Systems*, vol.**19**, pp. 1540-1552.

[7] Barre Jonathan, Rochange Christine and Sainrat Pascal (2008) Predictable Simultaneous Multithreading Scheme for Hard Real-Time, *Proc. of the 21st international conference on Architecture of computing systems*, pp.161-172.

[8] Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quinones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla (2013) A Hard Real-Time Capable Multi-Core SMT Processor, *ACM Transactions on Embedded Computing Systems*, to be published in 2013.

[9] The parMERSA project, http://www.parmerasa.eu/

[10] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar (2007) Utilizing dynamically coupled cores to form a resilient chip multiprocessor, in *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), pp. 317–326.

[11] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai (2001) Impact of CMOS process scaling and SOI on the soft error rates of logic processes, in *IEEE Symposium on VLSI Technology*, pp. 73–74.

[12] J. von Neumann (1956) Probabilistic logics and the synthesis of reliable organisms from unreliable components, in C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pp. 43–98.

[13] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb (1999) IBM's S/390 G5 microprocessor design," *IEEE Micro*, **vol. 19**, no. 2, pp. 12–23.

[14] D. Jewett (1991) Integrity S2: a fault-tolerant Unix platform, in *21st International Symposium on Fault-Tolerant Computing* (FTCS), pp. 512–519.

[15] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen (2005) NonStop advanced architecture, in *Proc. of the 2005 International Conference on Dependable Systems and Networks* (DSN), pp. 12–21.

[16] Y. Yeh (1996) Triple-triple redundant 777 primary flight computer," in *Proc. 1996 IEEE Aerospace Applications Conference*, pp. 293–307.

[17] E. Rotenberg (1999) AR-SMT: a microarchitectural approach to fault tolerance in microprocessors, in *29th Annual International Symposium on Fault-Tolerant Computing* (FTCS), pp. 84–91.

[18] S. K. Reinhardt and S. S. Mukherjee (2000) Transient fault detec- tion via simultaneous multithreading, in *Proc. 27th Annual International Symposium on Computer Architecture* (ISCA), pp. 25–36.

[19] T. N. Vijaykumar, I. Pomeranz, and K. Cheng (2002) Transient fault recovery using simultaneous multithreading, in *Proc. 29th Annual International Symposium on Computer Architecture* (ISCA), pp. 87–98.

[20] E. Schuchman and T. Vijaykumar (2007) BlackJack: hard error detection with redundant threads on smt, in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), pp. 327–337.

[21] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt (2002) Detailed design and evaluation of redundant multithreading alternatives, in *Proc. 29th Annual International Symposium on Computer Architecture* (ISCA), pp. 99–110.

[22] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomer- anz (2003) Transient-fault recovery for chip multiprocessors, in *Proc. 30th Annual International Symposium on Computer Architecture* (ISCA), pp. 98–109.

[23] J.C.Smolens,B.T.Gold,B.Falsafi,andJ.C.Hoe (2006) Reunion: complexity-effective multicore redundancy, in *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp. 223–234.

[24] M. de Kruijf, S. Nomura, and K. Sankaralingam (2010) Relax: an architectural framework for software recovery of hardware faults, in *Proc. 37th Annual International Symposium on Computer Architecture* (ISCA), pp. 497– 508.

[25] T. Bernard, K. Bousias, L. Guang, C. Jesshope, M. Lankamp, M. van Tol, and L. Zhang (2008) A general model of concurrency and its implementation as many-core dynamic risc processors, in *Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation* (SAMOS), pp. 1–9.

[26] Q. Yang, C. Jesshope, and J. Fu (2011) A micro-threading based concurrency model for parallel computing," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (IPDPSW), pp. 1668– 1674.

[27] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, and C. Jesshope (2012) Apple-CORE: Microgrids of SVP cores (invited paper), in *Proc. 15th Euromicro Conference on Digital System Design* (DSD).

[28] R. Poss (2012) *SL—a "quick and dirty" but working intermediate language for SVP systems*, University of Amsterdam, Tech. Rep. arXiv:1208.4572v1 [cs.PL], http://arxiv.org/abs/1208.4572

[29] R. 'kena' Poss (2011) *SL language overview*, University of Amsterdam, Tech. Rep., : https://notes.svp-home.org/sl15.html

[30] M. Lankamp and R. Poss, “SVP extensions to the Alpha ISA,” University of Amsterdam, Tech. Rep., March 2011. [Online]. Available: https://notes.svp-home.org/mgsim2.html

[31] L. Zhang and C. Jesshope, “On-chip COMA cache-coherence protocol for microgrids of microthreaded cores,” in Proceed- ings of the 2007 Conference on Parallel Processing (Euro- Par), 2008, pp. 38–48.

[32] T. D. Vu, L. Zhang, and C. Jesshope, “The verification of the on-chip COMA cache coherence protocol,” in Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, 2008, pp. 413–429.