On the exploration of the DRISC architecture

Yang, Q.

Citation for published version (APA):
Chapter 6

A real-time multi-core processor
—the extended study of DRISC

Abstract

One of the challenges for real-time multi-core processors is how all applications can get maximum benefits from the powerhouse while meeting real-time deadlines. This is the topic of much research targeting either task scheduling policies or hardware design. This chapter will investigate the existing DRISC architecture for real-time purposes. This is achieved through additional data structures and processes to support hardware priorities. The prioritized DRISC architecture inherits the original automatic concurrency management and data-flow scheduling offering different execution paradigms to both sequential and concurrent real-time applications. It manages to attain a balance between overall performance and timing bounds. This research is conducted in collaboration with the European Space Agency (ESA).

Contents

6.1 Background ................................................................. 108
6.2 The real-time strategy ...................................................... 110
6.3 Evaluation and analysis ..................................................... 114
Summary ............................................................................. 135
6.1 Background

Multi-core processors, either homogeneous or heterogenous, have already prevailed in desktops and servers to meet the ever increasing performance needs. A moderate tendency as well applies to embedded systems so as to act more efficiently on multitasking or emerging use cases possessing inherent concurrency, e.g. Cortex cores of ARM and Atom processors from Intel. Yet such a shift in the real-time field is not straightforward due to e.g. the painful migration of the “legacy” systems [NKN08].

The essence of real-time systems is to respond to the most urgent task and complete its execution either within the constrained period (hard real-time) or with mild tardiness (soft real-time), which requires not only functional correctness but also temporal correctness. For the sake of meeting a deadline, the core may devote all of its resources to that task until its completion in order to get rid of any interference from other tasks and then resume the interrupted job or jobs. However, this all-for-one strategy is quite inefficient on a multi-core, or even multi-threaded platform in the perspective of system throughput and resource utilization or energy consumption. On the one hand, there is no guarantee of full-time occupation of the privileged task over multiple cores; on the other hand some other job may be seriously delayed or violate its timing requirement. An alternative is sharing resources, usually at a core level. For example, make some cores real-time job exclusive and still allow other tasks to co-run on the remaining cores. If so, the concerns change as typically interference will be seen within other shared resources such as elements of the memory hierarchy, buses, networks etc. More aggressively, what if all kinds of tasks share all cores when plausible to acquire all computation power? Then the mutual interference stretches to pipeline and other shared resources on the same core. It is therefore a difficult trade-off in multi-core systems between real-time assurance and other metrics, which in turn challenges task management or even the hardware design.

6.1.1 Academic research

Studies on real-time task management of multiple cores chiefly involve introducing new or exploring suitable task scheduling policies based on existing algorithms like earliest deadline first (EDF), least laxity first (LLF) and rate monotonic (RM) [LL73] or proportionate fair (Pfair)[BCPV96, SA02], which by and large perform scheduling in either a global or a partitioned way. Global schemes on top of Pfair and its derivatives are known for high theoretical processor utilization (100%), but this ignores the preemption and migration overheads as well as degraded processor affinity [SHAB03]. Partitioned schemes are known for better scalability to multiple cores, whereas, they usually suffer a rather low utilization bound (only 50% if strictly partitioned) due to the trade-off between optimal partition and scheduling and their overheads. Other work like [AC06, CA08] stresses the processor affinity issue to make task scheduling cache-aware; [LRL09] manages to improve the utilization bound of the partitioned approach with fixed-priority preemptive scheduling. [LCC08, TSW10] propose hierarchical scheduling to enable co-execution of Real-time Task (RT) and non-RT. The former differentiates two types of tasks at the first level and applies LLF to the second level; the latter
increases scheduling diversity on each core by enabling various schedulers under
the control of the single RM scheduler by the OS.

Given the inefficiency when implemented at the software level from the field
of packet scheduling and packet multiplexing in real-time networks [SVCG04],
some literature focuses on implementing task scheduling in hardware as of way of
operating services making better use of hardware’s parallel nature. Such research
yields additional specialized hardware schedulers on multi-core chips. For example,
[JCWP05] proposes a co-processor scheme and offers the flexibility of switching
between hardware schedulers and software schedulers. The FPGA implementation
in [KSI03] provides three scheduling disciplines triggered and switched on the fly by
run-time requirements for better adaptivity. Gupta et al. in [GMM+10] presents
a Pfair hardware scheduler on Multiprocessor System-on-Chip (MPSoC) for lower
energy consumption and smaller scheduling delay; [NVS+12] makes a run-time
configurable scheduler for predictable performance of real-time applications and
[YMI07] introduces a real-time scheduler in charge of priority management and
poses IPC control on execution in addition to priorities to cater more for real-time
threads.

More radically, there have been proposals to design a multi-core architecture
with built-in abilities to handle real-time requirements using SMT cores, to exploit
both ILP and TLP. Cazorla et al. explore various real-time aspects on top of
such designs, e.g. [CKS+04] is one of enabling such an architecture to support
real-time demands and [CKS+05] describes how to perform scheduling. Mostly,
these goals are reached with operating system help. [UCS+10, PMM+13] construct
a multi-core processor capable of hard real-time, where each core is equipped with
dedicated hardware resources for RT, from stages of the super-scalar pipeline to
scratchpads for real-time data and instructions. Even the shared bus, cache and
memory controller are all biased or partitioned toward real-time needs in order to
guarantee the isolation of RT, and to ensure predicability and analyzability.

6.1.2 Motivation

One of the key features of the DRISC architecture is its ability to manage as well
as schedule the hardware threads using additions to the core’s ISA. This is the
equivalent of implementing parts of the operating system’s kernel in hardware and
compares to research leveraging additional on-chip hardware schedulers.

A major characteristic of the Micro Grid many-core system is the automatic
distributing and fair deployment of software parallelism to hardware micro threads,
giving a resource agnostic programming model. This together with the efficient
thread scheduling using data-flow synchronizers and hardware supported selective
barrier synchronization results in a very efficient architecture. If many hardware
threads are running on each core, it is possible to hide long latencies in instruction
execution and yield high-core utilization, even when code is distributed across
many cores. It is not uncommon for regular applications to achieve an average
utilization of 90% across even large numbers of cores.

An interesting question is whether the throughput-oriented DRISC multi-core
approach can be used for real-time applications or more precisely, is it possible
to retain higher efficiency while meeting stringent timing requirements. This falls
into two aspects:
Given more than one core, what impact does sharing resources such as cache and FPU have on RT allocated a complete core’s resources. Such a space sharing paradigm is common in studies of task scheduling on real-time multicore systems and we would like to investigate how the DRISC platform behaves without additional real-time logic.

Otherwise, is it possible to achieve a similar result by prioritizing a RT and continuing to timeshare it on the same core. Such a time sharing paradigm puts accent on benefiting all tasks with multiple cores. As an architecture for generic computing, either DRISC or Micro Grid does not prioritize RT of co-execution on the same core, since scheduling is fair and based on FIFO queues with dataflow scheduling (i.e. no resource locking) and frequent context switching. Resource allocation is based on FCFS and is non-preemptive. If any remaining resources are insufficient for a task, then the alternatives are to fail or wait. The other is that scheduling to the pipeline is fairly depending on the availability of data. We will address these problems while introducing the real-time components in the following section.

6.2 The real-time strategy

The uniformity of handling all kinds of tasks on the DRISC multi-core platform, e.g. the Micro Grid, makes it less competent to meet the timing bounds of real-time computations assuming tasks are sharing the same core. To achieve this, jobs abstracted as families of threads should be privileged according to their imminence and then RT can be specially catered for. To inherit hardware task management in hardware and avoid extra costs e.g. additional controllers, we would like to bring priorities into the DRISC architecture with support from current hardware data structures [BPNS12, NVS+12].

6.2.1 Resource allocation

The split-phase bulk creation has been already discussed in section 3.2.3 and shown in fig. 3.4. It comprises firstly reserving the minimal set of resources so that a family is able to run and secondly allocating resources up to a predefined limit specified in code. The key new components implemented in fig. 6.1 to achieve this biased resource allocation are the stand-alone queues 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 FCFS allocation only to requests of equal importance and allows threads with a higher priority to come last and still be served first.

[BPNS12] elaborates the globally shared priority queue with regard to the capacity, operations like filling and spilling, timing analysis and scheduling methods. It manages to circumvent the size and sharing limitations of the hardware structure to benefit applications via the help of software. While [NVS+12] puts emphasis on optimizations of the priority queue design to improve the performance of the associated hardware scheduler. For DRISC, the “priority queue” is actually a FIFO buffer and not shared among levels. The buffer size is moderate in terms
6.2. THE REAL-TIME STRATEGY

(a) Family allocation.  (b) Family creation.

Figure 6.1: Enhanced DRISC core to replace those in fig. 3.4 for bulk creation. Note in this and the subsequent diagram, multiple priority buffers are identified by the gray shadow to an existing queue.

of hardware costs concerning the content of each item unless there are too many levels to hold.

A crucial issue in resource allocation is to ensure successful contracting whenever a high-priority family request arrives. This can be guaranteed by resource reservation especially for higher priority levels under the principle that free resources retained for low priorities are subject to demands of the higher. A more flexible and dynamical way is applying a threshold to each level when there are a lot of levels in total [BRS08]. As to how many of each resource should be reserved or what is the suitable threshold for any priority level is a matter of system-level design, and it depends on the use cases for the target system. Note that both methods also pertain to resource disposition in family creation.

In this preliminary research we use only two priority levels\(^1\) (low and high as done in the transputer [HMSS87, dM92]), and instead of reserving or setting thresholds for resources we use an upper bound on configured resources which is greater than those required by any of the following experiments. This approach is possible in software simulation with the aim of determining what additional resources are needed to guarantee the instantaneous availability to different levels of priority for the applications tested.

6.2.2 Thread scheduling

The details of micro thread scheduling have been elaborated in section 3.2.4 chiefly upon data presence. With dynamic scheduling, high priority tasks could be biased with the support of extra queues shown in fig. 6.2.

\(^1\)The dual priority approach can satisfy ESA’s needs in space applications.
Figure 6.2: Enhanced DRISC core in comparison with fig. 3.6 for thread scheduling.

The proposed priority scheduling requires multiple active queues at the entrance of the pipeline so that the selected thread always comes from a non-empty queue with the highest priority. However the existing implementation moves all threads in a thread-neutral waiting list on the I-cache to the thread-neutral active queue and does so in a single cycle. Having one waiting list per cache line with mixed priorities would complicate such a shift when different priorities have been proposed. This would require a state machine to step through threads in the waiting list; it may also invert the priority serving them in a FCFS way to their respective active queue lists. Consequently, each cache line must also hold waiting lists for each priority level 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.

In addition to the I-Cache, threads might also be queued at registers due to missing operands. Nevertheless, 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 as stated in section 3.2.2. Only sibling threads of the same family may read the family’s global registers and they always have the same priority level on such an occasion. Therefore, the existing register file structure of DRISC is competent and no additional logic is required.

\[^2\text{It is in the very early stage of the created family that global registers might be waiting to be “pushed” by the parent thread. Only the first batch of child threads (up to the number of} \]
In addition to the above switching scenarios, the priority scheme introduces a slightly different speculation on labeled context switches. Normally when the active queue is empty, the currently running thread will not be switched as this switch is an optimization designed to avoid bubbles if data is not available in its registers, so it is better not to context switch in this situation. Nothing is lost as there are no other ready threads in the queue to fill the pipeline if switched. With priority queues, the same action is applied, even though an active queue of lower priority may not be empty. Although this may introduce a few unnecessary bubbles in the pipeline that could have been avoided, on the other hand it may avoid an unnecessary reschedule when there is only one high-priority thread running.

Finally a mild form of pre-emption is applied to the active queues. As the IF stage of the pipeline always keeps an eye on the status of 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 are always scheduled to run as soon as they are ready.

### 6.2.3 Priority management

Since resource allocation and thread scheduling have been modified to handle threads of diverse priorities, the next problem is how to associate each thread with its correct priority level so as to instruct DRISC cores to behave accordingly.

With respect to bulk creation, fixed priorities may qualify in the once-off resource allocation processes but what the optimal priority assignment is, is still a question. [Bar03] discusses such a problem based on the theorem of [ATB93] and formalizes it via the assignment-test proof. [KY07] investigates algorithms like direct priority mapping and shorter period upper to set the fixed suitable priority level to real-time threads. Our work simply suggests an explicit assignment while programming. The priority of a RT is exposed in code as part of the meta data in `create()` and will be used only at the root level in the concurrency tree. All sub-level families inherit this priority from their immediate higher-level parent, which in turn may inherit or be explicitly assigned if it is the root of a task.

As to dynamic scheduling, static priority might be insufficient in that it is pre-declared and does not take into account the real execution and the possibility of timing-bound violation, especially in a system of many priority levels as well as many kinds of running tasks. Although it is hard for the programmer to predict the safe priority level of a specific task, it is also unwise to assign the highest to all. As a result, the auto-tunable priority level according to the current execution progress against the proposed deadline is a natural choice, similar to EDF.

Unlike the conventional tuple timing-bound expression of a RT, the advisable deadline as expected execution time can be introduced in the same way as that of priority levels and stored in an additional field of each family entry. Then the key to “dynamic” priority is when to perform the deadline check to adjust priority. Instead of involving software efforts like [NVS+12] does or additional hardware investment while vacillating on the checking frequency, we would rather make h/w threads) may thence be suspended on these registers. They, at that moment, are of equal importance in either the static or dynamic priority scheme.
better use of the current data-flow scheduling, which proves to be fair and fine (empirically within few instructions), and carry out the work at dual granularity:

- The thread granularity. All threads of the same family mostly have the same behavior (SPMD) and are always set as equally important when created. As a result, the deadline of a family can be evenly divided and then allocated to each thread\(^3\). This demands an extra field in each thread table entry. While running, every thread has to check its own progress against its deadline immediately after rescheduling away from the pipeline each time. The thread’s tardiness will decide whether or not to change and how much to tune the priority level. Therefore an individual thread is entitled to adjust itself dynamically according to the instant situation and sibling threads are allowed to be weighed differently provisioning better flexibility.

- The family granularity. Given the iterative execution paradigm of exposed software threads and resource reuse policy (c.f. section 2.3.2). A family could do progress checking at each reuse point, that is when one child thread finishes its work and retires occupied resources, the family evaluates remaining workload against a set deadline to decide the priority level of the entire family, which then only affects subsequent concurrency mapping and may yields new micro threads of a higher priority level.

The procedure for either granularity can be smoothly merged into the current scheduling path of a thread with little cost, yet the validity and efficiency heavily depends on the accurate progress appraisal and the decision to adjust priority levels. This needs further investigation. Note that the initialized static priority is always regarded as the bottom line and the dynamic tuning would only make a positive increment.

To summarize, we can build a real-time multi-core processor on top of the existing DRISC architecture, where the resource allocation and thread scheduling are all implemented in hardware; therefore orthogonal to any operating system and software optimizations. We also rely on the existing data-flow scheduling to do these jobs without introducing new schedulers. Unlike affording rather many real-time specific components in \([UCS+10, PMM+13]\), we only add few buffers to implement hardware priority. We believe this together with HMT will make DRISC cores perform well in both execution and real-time efficiency. In the following part of the chapter will examine this.

### 6.3 Evaluation and analysis

In order to test and verify the proposed real-time strategy as well as our research targets, we have redesigned and configured the existing Micro Grid simulator as a quad-core system with two priority levels. The priority level or the number of priorities is a system-wide variable and is configurable in profiles, which conforms

---

\(^3\)There may be some use cases where only the deadline of individual threads makes sense. If so, the meta data value can be used to indicate this not the family’s, and thence the division here and the following family granularity are unnecessary.
6.3. EVALUATION AND ANALYSIS

to the current simulation pattern for flexibility. The difference from the specifications of the platform in table 3.1 are the 400MHz frequency (core, FPU and memory), 16KB L1 D-Cache, 256KB L2 cache and a single DDR2-800 controller\(^4\).

6.3.1 Aperiodicity with JetBench

First and foremost, we would like to examine the achievements of this new real-time multiprocessor through an aperiodic benchmark named JetBench. This is done on the basis of existing DRISC architecture and follows the conventional all-for-one execution pattern.

JetBench is a turbo jet engine simulator based on the models of NASA’s EngineSim application. It reads in flight profiles consisting of speed, altitude, throttle and deadline, performs thermodynamic calculations and yields a handful of parameters for three engine types. It is also friendly to multi-core platforms. More information of JetBench is available in [QMM10].

The JetBench source code\(^5\) piggybacks on OpenMP to parallelize the computational workload among scalable numbers of cores automatically. The pseudo code in fig. 6.3a shows how it does this. However this model is not supported in the DRISC platform. Instead we translate the parallel section into a family of independent threads, each of which does its computation from the input of one data point. The pseudo code of the DRISC’s concurrency version is shown in fig. 6.3b. This is equivalent to task level parallelism as each data point defines an independent task. Another change is that the deadline is deleted from the inputting quadruple but each task will report its own computation time, and the execution time of the entire family is also available. We then use both to compare results with those in [QMM10].

We fix the workload as 32 data points and vary the number of cores from 1 to 4 and hardware threads per core from 1 to 8, in contrast to [QMM10] that executes 30 data points on 1 ~ 16 single-threaded cores. Figure 6.4 shows the run time in seconds of each data point under every configuration. It is clear that the number of concurrent threads is decisive to the execution time of data points, as all curves are categorized into four groups according to the available hardware threads per core. The more threads, the longer run time given a fixed place size, and the disparities among different numbers of cores are increasing with increment of threads. Single-threaded cores yields the best results that all data points end around 0.15 second; two or four cores with 8 threads gives the longest time around 0.36 second. Note that even the worst case meets the 0.45 second deadline, which is scaled from the 9-second deadline on the 20MHz Simics simulator in [QMM10]. In other words, there are no points missing the deadline here, but the number of points with time violation in [QMM10] starts from 28 (1 core) down to the fewest 2 (8 cores) and then rises up to 26 (16 cores).

Since we have no predefined deadlines, the uniformity of all points is a critical factor reflecting possible miss rates. Figure 6.4 has already depicted this and together with standard deviations in table 6.1, we believe that any unevenness is due to competition for shared resources like FPU and memory. One or two

---

\(^4\)This is the reference configuration from the joint research with ESA and is based on the specifications of their new generation multi-processor (NGMP).

\(^5\)http://jetbench.sourceforge.net/
**Inputs:** Engine Type
Data File Defining Speed, Altitude, Throttle, Deadline

**Initialization:**
- Set Default Parameters
- Select Engine Type
- Open data file

**Parallel Section:**
- Calculate Pi
- Read an input data point

**Calculate:**
- Environment variables
- Thermodynamic parameters
- Engine geometry
- Engine performance

Print Results
If not EOF goto Parallel Section
Print Results
End

(a) Auto concurrency assisted by OpenMP [QMM10].

**Inputs:** Engine Type
Data File Defining Speed, Altitude, Throttle, Deadline

**Initialization:**
- Select Engine Type
- For each point in data file
  - Read input values
  - Set default parameters

**Parallel Section:**
- For each point as an independent thread
  - Calculate Pi

**Calculate:**
- Environment variables
- Thermodynamic parameters
- Engine geometry
- Engine performance

Print Results
End

(b) A family of independent threads.

Figure 6.3: The pseudo code of Jetbench with two-version concurrency.
6.3. EVALUATION AND ANALYSIS

Figure 6.4: The execution time in seconds for each data point. $A$\$B$ in the legend denotes the configuration as $A$ cores and $B$ h/w threads of each core.

Table 6.1: Standard deviation of 32 data points’ execution times in all scenarios.

<table>
<thead>
<tr>
<th>cores</th>
<th>h/w threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000011</td>
<td>0.0000014</td>
<td>0.00336</td>
<td>0.01136</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0000012</td>
<td>0.00211</td>
<td>0.01493</td>
<td>0.00042</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.0000012</td>
<td>0.004</td>
<td>0.0167</td>
<td>0.00127</td>
<td></td>
</tr>
</tbody>
</table>

threads per core pressures these elements only moderately creating rather trivial jitter; another extreme is that all related resources are saturated so that all threads are forced to share them “fairly” (e.g. 8 threads per core). This leaves the 4-thread configuration in the middle, which leads to a greater fluctuations in execution time. Another important factor is the iterative execution paradigm, where a thread executes provided there is a free hardware slot. As a result each thread faces a different system state or particularly shared resource usage. Yet such discrepancies are small most of time, shown by the fact in fig. 6.4 that threads of the same batch (physical concurrency per core) usually cost the same time. Note that all threads in this experiment are always of the same static priority level. This tells us that jitter may be mitigated by toggling dynamic priority regulation, but then the average execution time of all points under a specific platform configuration might be worse. We would like to revisit this in further studies.

When looking at overall performance rather than time to completion of an individual thread, the quad core platform is rather more sophisticated than that in [QMM10]. Figure 6.5 illustrates the execution time of the 32-data point workload and the system efficiency in terms of average IPC per core. Other than the increasing execution time of each data point, enabling larger parallel scale is an effective way to accelerate the whole task as 1\$1 gives the longest execution time of 4.8 seconds and 4\$8 the best of 0.36 second, though the advantage of multiple
cores shrinks along with increasing the number of concurrent threads, due to the decrease in efficiency of each core. Targeting 260 seconds with 30 data points, [QMM10] achieves a best time of 229 seconds with 4 cores and the worst of 272 seconds with 16 cores on the 20MHz Simics simulator, which in turn demonstrates the relatively better scalability in performance of this DRISC platform. Moreover, $\text{8}$ achieves a 90% busy pipeline manifesting the efficiency of dynamic scheduling. It may be even better by increasing the number of threads further in the single core case. The overlapped 2-core and 4-core efficiency curves tell that the system performance is primarily computation bounded due to sharing FPU between a pair of cores, and from 4 threads per core onwards the shared memory becomes another constraint. This would also help explain the diverse results across different numbers of threads in fig. 6.4.

### 6.3.2 Periodicity with synthetic benchmarks

We have designed two benchmarks to represent the spectrum of anticipated periodic RT. The first is the computation of square root (SQRT) by Newton iteration defined as $x_0 := x; x := (x_0 + y/x_0) / 2.0$ where $x$ converges to the $\sqrt{y}$. The SQRT is small, i.e. requires few processor cycles, and is sequential and computationally intensive (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 real-time applications.

The SQRT can run with different precision requirements which represent an linear increase in the number of iterations. However the result for 8-decimal place converged within 9 iterations and has already hit the upper bound of the double precision in our 64-bit system. There is no increase in accuracy beyond this.
6.3. EVALUATION AND ANALYSIS

Figure 6.6: Results of 16K FFT with various configurations (number of cores and hardware threads per core): average efficiency per core and speedup (dashed lines on the right secondary y-axis) vs 1 core/1 thread.

Therefore, we will limit the precision to the 8-decimal place in the following experiments indicating the heaviest computation load available from SQRT. Although iterations are logically sequential we could investigate its behavior by exploiting different numbers of hardware threads\(^6\)(1 and 2).

The second benchmark is a packet router task (ROUT) requiring a significant number of processor cycles. ROUT is parallel and memory intensive as the major part of it is a parallel copy of a message from an input buffer to the addressed output buffer. We believe that it is characteristic of many communication algorithms used in real-time applications.

Loosely based on the TCP/IP protocol, ROUT reads a packet header and extracts a 32 bit address which determines the output buffer required (1 of 4) to route the input packet. It then reads the packet length and copies the packet (from 20 bytes up to 64kbytes) in parallel from the input buffer to the addressed output buffer, i.e. each thread copies one byte.

6.3.2.1 Methods

In all experiments we run a RT periodically against the FFT, a long running background task. SQRT will execute at a nominal rate of once every 10K cycles (high duty cycle) and again at once every 20K cycles (low duty cycle), which with timing overhead is a nominal duty cycle of 10% and 5% respectively. We then choose a periodicity of 0.5M cycles (20% of duty) and 1.0M cycles (10% of duty) for ROUT. FFT is both computationally intensive as well as having non-local

\(^6\)if mapped to more than one hardware thread, each has to wait for the completion of its predecessor in sequence.
communication patterns, so it exercises all parts of the system, particularly the hierarchial caches (c.f. section 5.5.2 and table 5.3 for details).

To limit the exploration space, the precision of SQRT is restricted to 8-decimal place and the packet length of ROUT to 16kbytes. Experiments of space sharing are carried out sweeping through various parameters, namely number of cores and hardware threads for both RT and FFT, on the basis of which only typical corners are investigated in time sharing.

6.3.2.2 Basic Scenario

Figure 6.6 shows the results from the solo execution of the 16K FFT as a baseline. 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 to around 50% efficiency per core. For 1 and 2 cores efficiency is higher with a maximum between 80% and 90%. There is also a significant increase in efficiency between 4 and 8 threads (and hence speedup). We do see a marginal increase in performance with 16 threads, but given the additional use of resources it is not a good cost/benefit trade-off.

6.3.2.3 Space sharing

To explore space sharing, we define execution paradigms as follows:

- **Base** — RT threads are executed stand alone in the system but on one core.
- **Shared L2 Cache** — FFT is executed on a two-core place and the RT threads are executed on one of the remaining two cores that does not share an FPU with the 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 (Figure 6.6). This shows that the FFT is able to tolerate memory operations and the bandwidth limit on memory is not yet exceeded.
- **Shared FPU** — FFT is executed on one core and the RT threads are executed on the other core sharing the same FPU.

In the above scenarios, prioritizing RT is unnecessary as the real-time strategy implemented on the DRISC platform only has a core scope, but FFT and RT never run on the same core.

The jitter in the execution of RT as well as FFT are depicted in figs. 6.7 to 6.9. In no case do the FFT and RT disturb each other a lot except for the cold start points, where SQRT suffers a jitter of 25% ~ 30% (fig. 6.7) and ROUT from 5% in sharing FPU to 10% in sharing L2 cache (fig. 6.8). We believe the anomaly here is due to the interference from memory operations of FFT during family creation (reading instructions from memory) of RT. It is the queued memory requests that add significantly to SQRT compared to short execution time but only a little to the longer-running ROUT. The subsequent points from cyclic execution show the feature of RT very well besides its good performance. The computation-intensive SQRT only has jitter when sharing the FPU.
Figure 6.7: Jitter of SQRT as a percentage of the baseline execution time. RT runs periodically and data is collected during FFT’s execution. In the legend $x$@$y$ represents the hardware threads allocated to FFT and RT respectively (this is followed by all figures afterward otherwise specified).
Figure 6.8: Jitter of ROUT as a percentage of the baseline execution time. ROUT follows the same execution pattern of SQRT.
6.3. EVALUATION AND ANALYSIS

Figure 6.9: Performance of FFT across a range of parameters relative to each matched baseline value in fig. 6.6.
On the other hand the curves for memory-intensive ROUT fluctuate a lot because of the heavily-stressed memory when sharing L2 cache, e.g. 8 hardware threads of FFT. By and large the fewer hardware threads the FFT uses, the better memory locality and the better RT results, which is especially true for the memory intensive ROUT. The analysis also pertains to the execution time of the FFT, for example it suffers trivial penalties when sharing L2 cache with ROUT (fig. 6.9b) and sharing FPU with SQRT (fig. 6.9a) but is slightly accelerated in most cases particularly the other sharing mode with each RT. Note that all data applies to both high duty and low duty patterns as they give the identical results.

6.3.2.4 Time sharing

Using time sharing mode enables all kinds of tasks to benefit from multiple thread and cores. To narrow the research space, we only choose the best performance configuration according to the empirical space sharing results, i.e. 8 threads for FFT, 2 threads for SQRT and 4 threads for ROUT, and run them in the following patterns:

- $1+1$ — FFT and RT both execute on the same single core.
- $4+1$ — FFT takes all 4 cores on one of which RT executes.
- $4+4$ — FFT and RT share all 4 cores (ROUT only).

In these experiments, FFT and RT always share the same core or cores and the real-time strategy comes into use.

Figures 6.10 to 6.11 show the jitter of both prioritized and non-prioritized RT in the time sharing mode. The issue here is the benefit from the implemented real-time strategy, approximately a 55% reduction of jitter. The $1+1$ scheme always yields the best result and $4+1$ the worst. It is clear that the final results depend much on how the FFT is treated given the biased RT. When FFT is running on one core, its activities, either arithmetic computations or memory accesses, are well contained by the prioritized thread scheduling so that the RT is least affected. On the contrary, running the prioritized RT only on one core while the FFT runs as usual on the other three cores is less helpful on account of the already saturated memory bandwidth (fig. 6.6). $4+4$ improves the situation a bit but is still inferior to $1+1$ because it is not possible for the thread scheduling on four cores to follow the same tempo. In a word, the better FFT is regulated by hardware priority, the lighter pressure on the memory system and the lower jitters of RT. As to the advantage of high duty over low duty, although the shorter period leads to more iterations and operations, it may retain cache locality better.

More statistical information is revealed in fig. 6.12. If evaluated by the balance of three bars in each group, $1+1$ with priority is still the only candidate that gives significantly better results when compared to running the unprioritized RT. However in fig. 6.12b, even the maximum of $4+4/P$ is only 48% of the minimum of $1+1/P$, which proves the gain from multiple cores. One interesting thing in fig. 6.12c is the 5% ~ 8% performance improvement of FFT under both $4+1$ and $4+4$ patterns with pure data-flow scheduling, albeit with the memory bandwidth limitation. This is only because of the co-running RT that disturbs the scheduling of FFT, especially its memory requests. The total amount of memory transactions might have slight changes but the memory concurrency of FFT is well regulated
Figure 6.10: Jitter of SQRT as a percentage of the baseline execution time. The high duty and low duty data is collected only during the FFT’s execution; dashed curves are the results of non-prioritized RT and dotted lines with numerical labels are the average levels of each curve (this is followed by all figures afterward otherwise specified).
Figure 6.11: Jitter of ROUT as a percentage of the baseline execution time. ROUT follows the same execution pattern of SQRT.
and the contention for shared resources (i.e. bus and L2 cache) is alleviated by scheduling the RT. Both are of great importance to performance as discussed in chapters 4 to 5. This is also proved by the actual results that the co-running of ROUT is better than that of SQRT, 4+4 is better than 4+1 and high duty is better than low duty.

6.3.2.5 Space sharing vs. time sharing

Putting all the above together, space sharing offers the most stable results for RT execution but in these experiments the FFT was only ever run on 2 cores and as provisions does not saturate the shared memory resources. However as we have argued, the overall performance is another key factor, provided the timing issues are not compromised.

We have seen how much ROUT benefits from running multiple cores in fig. 6.12b. These values of FFT shown in fig. 6.13a also conclude that the more cores, the more gains, but this is inevitably affected by the RT. It is also true that the less curbed by RT (prioritized or not), the better results FFT can yield, e.g. 4+1 and 4+4. Ditto to processor utilization in fig. 6.13b as the FFT is the chief contributor. It is also worth noting that the scheduling of DRISC always tries to approach the 100% optimal utilization as long as asynchronous delays can be hidden as far as possible, which makes the dynamic scheduling of DRISC advantageous over the global and partitioning methods. Nevertheless the sharing L2 cache mode is highlighted in both cases when taking RT jitter into account, but 4+4/P of ROUT excels in promoting its performance (execution time) a lot.

All in all, space sharing implies always executing normal tasks conservatively (i.e. by reserving cores) in case of break-in RT at any time and it wastes the computation power. This may be a good solution for sequential tasks like SQRT, but for RT with inherent concurrency, time sharing, e.g. 4+4/P, enables better performance with moderate jitter and the full use of all cores. Meanwhile, both space sharing and time sharing share one problem in common, the interference from sharing unprioritized resources between the RT and FFT. We will look at this in the following section.

6.3.2.6 The shared data path

The proposed hardware priority gives privilege to RT during concurrency creation and scheduling, but it still leaves some other shared resources priority neutral and in particular the data path from D-Cache to the chip boundary. The consequence of this can be seen from previous experiments, e.g. the anomalies of cold start in space sharing (figs. 6.7 to 6.8) and a little higher jitter in time sharing (figs. 6.10 to 6.11). We ascribe these results to the high levels of parallelism in the background task that applies great pressure on the unprioritized shared data path. Although we have addressed in previous chapters that such a path (except the D-Cache) is beyond the critical path and the access latency can be hidden via multithreading, yet only if the required data is available, can the real-time threads be chosen to execute by the data-flow scheduling in a prioritized manner. As a result, further accelerating high priority tasks to minimize co-running jitter can be achieved via:
Figure 6.12: Statistical results of all benchmarks. Base levels are shown in dotted lines where 1 or 4 in brackets denotes the number of cores. In all legends and labels, */P represents RT is prioritized, otherwise not (this is followed by all figures afterward if not specified).
6.3. EVALUATION AND ANALYSIS

(a) FFT’s performance: normalized to the baseline FFT of 1 core/1 thread.

(b) Processor utilization: the average of working cores.

Figure 6.13: Overall performance: results of space sharing are limited to task configurations of time sharing. In legends, either “baseline” or “FFT_only” indicate the baseline results of FFT in fig. 6.6.
Figure 6.14: Jitter of ROUT as a percentage of the baseline execution time of 4 cores/4 threads. This is made by varying the concurrency of FFT denoted by A$B (A core/B thread) in the legend.
• prioritizing the shared data path. This can be made either following [PMM*13] that offers dedicated caches and connections to RT, or continue the current hardware priority strategy to provide dedicated buffers. The former may yield better results at the expense of circuits and scalability (to cater for different priority levels) while the latter is just the opposite;
• suppressing normal tasks in the background. If the scale of (memory) concurrency is well contained, for example no more than 4 threads in the FFT (the number might be even lower in the 4-core case), extra memory bandwidth will be released to RT (ROUT in particular) for better performance and this may impact the FFT slightly.
• improving available bandwidth budget. Other than sacrificing the performance of the background task, higher bandwidth makes it possible that even when the FFT puts pressure on the shard data path (as in previous experiments), there is extra memory bandwidth for RT to moderate its jitter.

Apparently the first solution needs more effort on the architecture side and we would like to defer that for future research. Now we just examine whether and how the other two schemes work.

Concurrency control

On account of the jitter in different execution paradigms, we choose time sharing and ROUT, and run the FFT in two ways:
• 2$4 — FFT executes on 2 cores each with 4 threads.
• 4$2 — FFT takes 4 cores each with 2 threads.

but fix the prioritized ROUT with 4 cores/4 threads in both low duty and high duty. We also take the 4 cores/8 threads (4$8) results from previous time sharing experiment (shown as 4+4/P in figs. 6.11 to 6.13) for comparison.

Figures 6.14 to 6.15 demonstrate the achievements of dampening the concurrency scale of FFT. The ROUT’s execution jitter of either high duty (fig. 6.14a) or low duty (fig. 6.14b) shows that reducing FFT’s overall parallelism does benefit the ROUT. The regulation can be achieved through tuning the hardware concurrency per core, the number of exploited cores or both. However using fewer cores is more effective as 2$4 brings only 12% overheads (jitter) in average to the ROUT in comparison with 20% of 4$2. Both are much better than previous 4$8. Moreover, the penalty to FFT’s performance in table 6.2 resembles the variation of RT jitter. It may expect even lower overhead e.g. in 1$1, whereas this should always take the absolute performance of FFT (fig. 6.6) into consideration to make trade-off.

The similar changes can also be seen from the processor efficiency shown in fig. 6.15. Although the fewer threads or cores, the more stable utilization rate, yet neither is better than the baseline situation of FFT. This is because of the prioritized scheduling in following two aspects:

• FFT is dominant, as its instruction number is 2 orders of magnitude more than ROUT’s\(^7\), but its execution is always curbed;

\(^7\)The periodical execution of ROUT has only 25 (high duty) or 12 (low duty) iterations around in total during the execution of FFT.
Table 6.2: Performance penalty of FFT when controlling its parallel scale.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Penalty</th>
<th>High duty</th>
<th>Low duty</th>
<th>Actual performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2@$^4$</td>
<td>2.64%</td>
<td>0.69%</td>
<td></td>
<td>4.042</td>
</tr>
<tr>
<td>4@$^2$</td>
<td>13.52%</td>
<td>12.5%</td>
<td></td>
<td>4.265</td>
</tr>
<tr>
<td>4@$^8$</td>
<td>25.82%</td>
<td>24.33%</td>
<td></td>
<td>4.287</td>
</tr>
</tbody>
</table>

N.b. The percentage is relative to the matched baseline value in fig. 6.6. The performance is the average speedup (vs. baseline FFT of 1 core/1 thread) of the high duty and low duty.

Figure 6.15: Processor utilization rate as average IPC per core. In the legend “FFT_only” indicates the baseline value of matched configuration shown in fig. 6.6. Note that such a value used in 2@$^4$ is scaled (divided by 2) to 4 cores.

- data-flow scheduling manages to fill the pipeline when it used to be empty or idle during the solo execution of FFT to improve efficiency, but always prioritizing RT may introduce extra bubbles and stalls.

Nonetheless, such an impact is moderate when the shared data path is far from saturation, e.g. 2@$^4$ and 4@$^2$ cases.

**Higher bandwidth capacity**

As an alternative approach to conquer the bandwidth problem, we empirically increase the capacity by adding another DDR2 controller (channel) or using DDR3 of higher frequency\(^8\), on account of the fact that the on-chip connection is rather

---

\(^8\)Each DDR2 channel provides max. bandwidth of 51.2 GB/s while DDR3 provides 153.6 GB/s.
6.3. EVALUATION AND ANALYSIS

(a) Jitter of ROUT as a percentage of the baseline execution time of 4 cores/4 threads.

(b) Performance of FFT (normalized to the baseline value of 1 core/1 thread in fig. 6.6).

Figure 6.16: Results for higher bandwidth. In legends and labels, “hd” and “ld” indicate “high duty” and “low duty” respectively.
simple here and the pin bandwidth is very likely to be the main bottleneck. According to results in fig. 4.3, this would effectively boost performance in the context of larger parallel scale. We then repeat the experiment of 4+4 in time sharing and compare with those results.

With one additional DDR2 channel, the baseline FFT’s performance of 4 cores/8 threads rise to 0.855 in average IPC per core, and DDR3 gives 0.7187. Although this approaches the saturated bandwidth, very similar to the 4+4 situation (saturated but at 0.5), we still get very appreciable jitter for RT, which then is quite different from the 4+4. As shown in fig. 6.16a, the jitter of DDR3 is around 6% (10% from DDR2x2) as the benefit of the extra bandwidth. This is a sharp contrast to the values of 4+4 and even surpasses concurrency controlling by a factor of 50%. It is also notable that the impact from execution duty is diminishing with increasing bandwidth.

On the FFT side shown in fig. 6.16b, DDR2x2 renders 71.28% better performance than the baseline result while DDR3 44%. This could be credited to the memory access of FFT that usually switches between different rows, and striding over two controllers could alleviate DRAM’s tension. However offset by the prioritized scheduling, both almost drop down to the same level, which are still better than the 70% loss of 4+4. As we have analyzed before, the FFT dominates the system performance and the decrement of its performance augurs badly for the overall performance. This is illustrated as the processor utilization rate in fig. 6.17. DDR2x2 and DDR3 again give nearly the same result, around 23.4% higher than that of 4+4. This may be not very exciting for the chip efficiency, but it inversely proves the success of the proposed priority scheme.

---

9 We also explore to provision larger caches. However only with enough pin bandwidth can applications benefit from larger caches, otherwise there is only a negative impact.

10 Both reach the maximum with 11 threads. DDR3 has a value of 0.7444 and dual DDR2 0.869.
To wrap up, with a fixed hardware budget, suppressing the (memory) concurrency of the background task provides a relatively easy and convenient way to ameliorate the shared data path and then the jitter of RT. More importantly it is proved to be effective. As long as it is carefully tuned, neither the actual performance of background back nor the processor utilization rate will be greatly impacted. Increasing pin bandwidth is another alternative of hardware optimization. It not only moderates the jitter of RT, but also helps achieve better performance for the background task and efficiency for the processor. Additionally, it reaffirms previous study in section 4.4.2.

**Summary**

Meeting deadlines is the major concern of real-time processing but system efficiency is also important due to the energy issue. It is not difficult to make one outstanding over another but is hard to make both. The DRISC architecture, with its in-silicon concurrency management and data-flow scheduling, can better the processor efficiency by just reserving thread slots not cores for RT. Nonetheless it is not good at supporting both non-RT and RT in a time-multiplexing way on the same core while guaranteeing timing requirements. Because of this, we have introduced hardware priority into DRISC and reinforced the concurrency management and scheduling to the benefit of RT, so as to build a DRISC multi-core processor that supports different execution paradigms for the real-time purpose. The experiment of an aperiodic benchmark in the all-for-one paradigm shows the advantages in per-thread and overall performance of the quad-core system. However it also stresses interference between threads of equal importance and implies the importance of dynamic priority adjustment. For periodic RT, two different execution paradigms namely space sharing and time sharing are investigated on how to make better use of multiple cores in the real-time field. Our experiment shows that concurrent RT can be greatly accelerated by sharing all cores with reasonable jitter, and sequential codes will get better timing results when executing on an exclusive core. We also noticed the caveat that the shared data path tends to be the bottleneck of both execution time and jitter, especially in the time sharing paradigm with too much memory pressure. We later verified that meticulously regulating the (memory) concurrency of background non-RT could effectively moderate this while keeping satisfying results; investing in hardware to break such a bottleneck makes everything even better. Further architecture refinements toward this problem for more stringent timing bounds as well as how to support more priority levels over even more cores is the key of our continuing research.