On the exploration of the DRISC architecture
Yang, Q.

Citation for published version (APA):
Yang, Q. (2014). On the exploration of the DRISC architecture

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Review and future

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Review and conclusion</td>
<td>140</td>
</tr>
<tr>
<td>7.2</td>
<td>The next steps for this research</td>
<td>144</td>
</tr>
<tr>
<td>7.3</td>
<td>Other DRISC related research and issues</td>
<td>148</td>
</tr>
</tbody>
</table>
7.1 Review and conclusion

To investigate a scalable multi-core system, the CSA group initiated system-level research using many DRISC cores, which has yielded the Micro Grid and its software simulator. Based on the existing achievements, the work presented in this thesis has two goals:

- to understand and rectify some of the scalability issues that have been identified in the prior evaluation of the Micro Grid;
- to extend the evaluation into the real-time domain and to redesign the DRISC core to enhance its efficacy there.

7.1.1 Traffic regulation for performance scalability

Previous studies show that the Micro Grid system performance is easily saturated or even degraded as hardware concurrency is increased beyond a certain threshold. This is understandable when increasing physical concurrency by adding cores to the application, as this increases the load on shared resources such as the memory system. However, we also see an impact on performance when increasing the number of hardware threads, which does not increase the number of memory operations issued by the cores involved. Theoretically assuming sufficient concurrency, thread interleaving can hide delays from asynchronous operations, keep the pipeline busy and deliver an efficient system with very few pipeline bubbles or stalls. Typically on well behaved code we see pipeline slot utilization of up to 90%. Nonetheless, when we observe scalability issues, the accompanying rather low core efficiency tell us that some delays cannot be tolerated with massive concurrency, which suggests the memory system becomes saturated as we increase the number of threads.

Experimenting on a 128-core chip, we noticed that a benchmark with 20% memory load (measured as memory operations in percentage to the total instructions) and good locality can achieve 16% higher performance from an investment of 20% more on-chip bandwidth, though the average core efficiency is still lower than 20%. In contrast, using larger caches and more memory controllers brings nearly no improvement (<1%). This identifies the on-chip network as the potential source of the bottleneck in the memory system. Since the interconnect frequency cannot endlessly be scaled up, the question is what else can we do given a limited bandwidth or hardware budget? What is the root cause to the saturation on some applications when we increase the number of threads? Remember there are no additional memory instructions being issued although their scheduling will change as we increase the number of threads. To answer these questions, we started from the network payload to investigate how the bandwidth is used or depleted to saturate performance.

By monitoring on-chip traffic at the memory controllers, we recorded how memory network transactions scale with hardware concurrency (from either more cores or physical concurrency per core), and noticed how the quantity of transactions is closely coupled with the performance variation. With these heavy payloads, the network is easily overloaded leading to saturated performance. As a result, reducing the on-chip memory traffic seems a reasonable strategy to adopt.
On further inspection of the traffic composition, we found it is the coherence transactions resulting from correlated memory writes that chiefly contribute to the traffic explosion. Both their quantity and proportion to the total traffic often rise along with the number of cores or hardware threads per core. The question then is how to deal with memory writes to lower the coherence overhead and a solution to this is key to the traffic problem.

There are existing software techniques to reduce memory write instructions when some stores are "redundant", whereas this is not the whole case in our benchmarks. In order to achieve the expected results, we manually reorganized the write granularity to cut the number of stores down to one eighth of the original code per thread. We showed that this works in decreasing the traffic load, however it also gives us very inconsistent results and in some cases even reduces performance (as shown in fig. 5.1). Besides, the refactoring is non-trivial and its major shortcoming is that not all applications that suffer such a traffic problem can be tackled in the same way.

Other than the non-trivial redesign of a new coherence protocol or memory subsystem, we want to find a solution with simplicity, better scalability, flexibility, generality and cost efficiency. The key is that the scheduling of many hardware threads seems to increase the number of transactions in the network. Given the coherence protocol used in the Micro Grid, it is reasonable to assume that this increment in traffic means the transactions are occurring at a finer level of granularity, i.e. fewer thread writes per transaction. Hence we adopted the philosophy of write merging, and implemented the combination logic WCB as a part of the DRISC architecture. It is customized for the DRISC core by sharing a similar structure and the same data path with the D-Cache to minimize hardware costs, and thus is different from others in both implementation and management. We then verified this scheme and explored the design space, i.e. the capacity and merging granularity, from several benchmarks featuring different concurrency scales, memory density and store locality. From the results of this investigation we found that:

- the WCB can effectively reduce coherence transactions from on-chip traffic;
- the reduction of traffic always yields an improvement in system performance. It brings an average of 36.89% improvements to all referred benchmarks, in particular 169.1% for the previously mentioned benchmark;
- the performance achievements from write combination, results from not only fewer memory transactions in the network, but also looser temporal distribution of the traffic with reduced number of stalling cycles observed (fig. 5.7). However the quantitative change in on-chip traffic dominates the qualitative variations;
- the WCB with a moderate size (one or two entries) is usually sufficient to achieve the optimum improvement and thus bears good cost efficiency;
- memory ordering imperatives might alleviate the memory pressure for synchronization by flushing WCB and thus yields further performance;
- the WCB has better cost efficiency than using larger caches when dealing with such on-chip network traffic issues. It is also useful to mitigate the off-chip bandwidth pressure without increasing the number of pins.
Given the achievements from WCB and other spin-offs during this research, we can reach the following conclusions:

- write combination is a cheap but effective way to control coherence traffic from correlated writes, thus it supports the scalability of hardware coherence;
- write combination provides an effective method of alleviating on-chip network saturation from coherence traffic explosion when using a large number of threads. The same idea is also exploited in case of hot spots in the interconnect network, e.g. Cray’s XMT [KV11, STV12]. However it has to work with other memory system optimizations to break the existing bottleneck, for example holding back the update transactions in the combining buffer might increase the synchronization overhead;
- write combination indicates the importance of memory concurrency management. Both efficient controlling and scheduling are crucial to the on-chip bandwidth issue. All techniques for this purpose would possess better applicability and adaptability, and promote system scalability further.

### 7.1.2 Hardware priority for real-time demands

To utilize multiple cores for real-time processing, the scheduling strategy always plays an important role in meeting the timing requirements of RT. However current approaches are not usually very efficient as least in two aspects:

- typical policies lead to low processor utilization rate [SAA+04, LRL09], i.e. global scheduling that offers processor-exclusive execution has to suspend other tasks and cannot guarantee full-time occupation of the privileged task over multiple cores; while partitioned scheduling that allocates cores among tasks with different criticality imposes compromised execution. This is detrimental to making full use of a multi-core processor;
- its implementation in software possesses good flexibility, whereas it is also known to fail in case of extreme urgency because this usually involves operating systems and bears longer scheduling delays. A hardware scheduler may work better, but the additional logic still cannot always guarantee instantaneous responses.

Given the existing DRISC architecture, it is easy to achieve the execution paradigms of both global and partitioned scheduling by declaring specified resource requirements when making concurrency creation. The in-silicon concurrency manager can serve a request as soon as it arrives and the run-time scheduler can push a thread into execution as soon as its code is in I-Cache, which minimizes the scheduling delays. During the study of Micro Grid, we often see very high DRISC core efficiencies in a system with moderate numbers of cores. This is also shown in the processor-exclusive execution of an aperiodic real-time benchmark (figs. 6.4 to 6.5).

However, more aggressively, can we time-multiplex all tasks on all cores instead of partitioning a processor for each of them? Or in other words, is it possible to benefit RT from all cores under a general workload by partitioning thread slots, while meeting deadlines and maintaining the merits of the architecture? Although
it is easy to achieve such a time-multiplexed paradigm in the existing DRISC, accomplishing real-time deadlines is another matter because both concurrency management and thread scheduling do a very fair job and RT can never be privileged. The result of this is that any single task executing as a thread in a DRISC core will have an execution time that is inversely proportional to the number of thread slots it shares the core with, which is counter productive. Consequently we introduce hardware priorities and associate more critical tasks with higher priority levels. The hardware cost of this scheme comes from additional dedicated FIFO buffers matching the number of priority levels, while resource allocation and thread scheduling also have to be tuned for prioritization. These buffers are simply pointers into the thread table and are negligible compared to the major structures in the core such as cache, register files and thread and family tables.

The DRISC with hardware priority helps achieve thread slots partitioning (time sharing), from which a parallel RT can run 2.797x faster in comparison with core partitioning (space sharing) while the background workload gets almost the same actual performance. More precisely

- RT have very stable execution with an exclusive core on account of the low jitter observed. This is a good news to sequential tasks, but parallel RT have to sacrifice the full-speed of acceleration from multiple cores;
- when time multiplexing tasks with mixed criticalty on the same core, the proposed strategy provides privilege to prioritized tasks well, and contains the execution time jitter to a rather low level (compared with non-prioritized execution);
- what we observe when sharing all cores with both background and real time tasks is an increase in execution time jitter, but we have identified this problem as one of saturation of the shared resources. The configuration of the platform was based on the NGMP multiprocessor, which uses a non threaded core. With the more efficient DRISC core, the bus/memory bandwidth becomes saturated when running demanding code on all four cores, and any final design needs to have its shared resources provisioned so as not to limit execution time;
- the background task may also benefit a bit from time-multiplexed execution because the scheduling of real-time threads mitigates its peak hit to shared resources.

We verified that either controlling the memory concurrency of the background workload or increasing off-chip bandwidth is useful to ameliorate situations where we see shared resource saturation. This improves real-time results of the time-multiplexed execution significantly. Meanwhile, this also reaffirms the resource sharing or bandwidth issues discussed in the many-core research.

Generally, this preliminary research has promised the possibility of building a multi-core real-time processor on the basis of DRISC architecture to support mixed criticalty. The processor not only retains the scheduling and execution efficiency of DRISC, but also accelerates all applications using all cores through time-sharing, where thread slots instead of cores could be reserved for more important tasks so as to improve system efficiency and processor utilization rate. However the full accomplishment of this needs investigations of a number of issues that are not or only slightly discussed in this thesis, for example
• in section 6.2.1 we talked about how to allocate resources, namely family and thread table entries and registers, but did not examine this in later experiments. Instead we offered sufficient resources for the benchmarks running concurrently. How to allocate thread resources (hardware table and register file entries) is very much application dependent, as is the number of priorities required. All of our experiments were performed with just two levels of priority, namely one for the real time tasks and one for the background tasks. Indeed our partner in this work, The European Space Agency, believe that such a configuration would be sufficient for their requirements. However we note that reserving thread slots on all cores for priority is a trade off. If we maintain a full set of threads for the background tasks then this does not impact the performance of the background workload at all. However, there is a significant cost in reserving additional table and particularly register file entries for the reserved thread slots. Any design will be a tailored compromise between requirements of both background and RT deployed, and the silicon budget;

• in section 6.2.3 we talked about dynamic priority but did not investigate this. Also we used jitter and not time violation for evaluation. The key to the more general problem is how to define deadlines suited to the requirements of all kinds of applications and then how to evaluate “progress” towards those deadlines with the use of dynamic priority. These problems must be explored before moving forward in this direction.

Apart from any DRISC improvement, system level efforts are still needed for more strict timing bounds and better scalability. In addition to the existing enhancements to DRISC, it could be beneficial to ensure that data for real-time threads is delivered with more expediency, which could further accelerate task execution and give better timing results. This requires privileged data path(s) for tasks with diverse criticality and is likely to incur scalability issues. Similarly, affording more cores within a processor exacts sophisticated system design and should learn lessons from previous research for scalable systems.

7.2 The next steps for this research

Section 7.1 has summarized the chief work of this thesis, and has justified the adopted solutions and schemes to handle specific problems. Although they are helpful to approach our research targets, there is still extra work either to perfect the current strategy or to try alternatives to attain possibly even better achievements.

7.2.1 Scalable memory

We have presented existing memory subsystems for the Micro Grid in section 3.3.2. DCR and the its variation of the two-leveled memory ring are the current memory candidates for the DRISC many-core architecture. We have investigated each in chapter 5 and [YFPJ14] on the traffic problem which brings a penalty to system performance. The negative impacts might partially be credited to the limitation of the ring structure that forces memory transactions to travel all around the
interconnection, which leads to poor scalability, as for global communication the bandwidth per node is inversely proportional to the number of nodes. Given the openness of the Micro Grid to any memory type, we would like to promote the scalable memory subsystem research by trying different structures.

Figure 7.1 is the one under discussion which arranges distributed caches into a 2-D mesh instead of ring and each L2 is positioned by its coordination. For the interest of coherence among L2 caches, it is impractical to retain message broadcasting to update non-exclusive memory blocks like DCR does, and only the sharing set of cache lines is essential for scalability. One way to realize this is associating each memory block to a host L2 cache which can be traced directly by memory addresses. Hosts are agents for communicating with memory controllers for off-chip accesses, recording sharing information in directories and maintaining coherence. A local L2 miss will be routed to the host first and then served accordingly. Possible actions are listed in table 7.1 using invalidation as the example. However the challenge there is how to ensure ordering. Although all coherence messages are sequenced at the host L2 cache, the host cannot guarantee the receiving order is as strict as they are emitted, which is especially detrimental to
Table 7.1: Overview of coherence maintenance in the new memory system with invalidation.

<table>
<thead>
<tr>
<th>Local event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read hit</td>
<td>nop</td>
</tr>
<tr>
<td>Read miss</td>
<td>request cache line from host.</td>
</tr>
<tr>
<td>Write hit</td>
<td>shared notify host to invalidate others and then commit data.</td>
</tr>
<tr>
<td></td>
<td>exclusive commit data.</td>
</tr>
<tr>
<td>Write miss</td>
<td>request cache line from host and then commit data.</td>
</tr>
<tr>
<td>Eviction</td>
<td>shared drop and notify host.</td>
</tr>
<tr>
<td></td>
<td>exclusive write back to DRAM and notify host.</td>
</tr>
</tbody>
</table>

(a) Actions for local events: local.

<table>
<thead>
<tr>
<th>Local event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read hit</td>
<td>nop</td>
</tr>
<tr>
<td>Read miss</td>
<td>check directory to confirm; grab from DRAM if not cached or from remote; register or change in directory.</td>
</tr>
<tr>
<td>Write hit</td>
<td>shared invalidate others and then commit data.</td>
</tr>
<tr>
<td></td>
<td>exclusive commit data.</td>
</tr>
<tr>
<td>Write miss</td>
<td>check directory to confirm; grab from DRAM if not cached or from one remote and invalidate all then commit data; register or change in directory.</td>
</tr>
<tr>
<td>Eviction</td>
<td>shared write back to DRAM and deregister in directory.</td>
</tr>
<tr>
<td></td>
<td>exclusive</td>
</tr>
</tbody>
</table>

(b) Actions for local events: host.

<table>
<thead>
<tr>
<th>Remote event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data request</td>
<td>send data to the remote requestor.</td>
</tr>
<tr>
<td>Invalidation</td>
<td>invalidate the local copy and acknowledge if required.</td>
</tr>
</tbody>
</table>

(c) Actions for remote events: local.

<table>
<thead>
<tr>
<th>Remote event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Miss</td>
<td>check directory to confirm; grab from DRAM and redirect to remote if not cached or notify a remote transfer; register or change in directory.</td>
</tr>
<tr>
<td>Invalidation</td>
<td>notify to invalidate all the other copies.</td>
</tr>
<tr>
<td>Write miss</td>
<td>check directory to confirm; grab from DRAM and redirect to remote if not cached or notify a remote transfer with invalidation; register or change in directory.</td>
</tr>
<tr>
<td>Eviction</td>
<td>deregister in directory.</td>
</tr>
</tbody>
</table>

(d) Actions for remote events: host.

memory updates. Conquering this problem mostly relies on either an ordered interconnect or an ordering scheme over unordered interconnection network. DCR possesses a good ordering property while some additional work has to be done on
7.2. THE NEXT STEPS FOR THIS RESEARCH

this new mesh network. [Sub13] may be beneficial that recently realizes a global
ordering scheme over a 2-D mesh network for snoopy coherence in a 36-core chip.

To further simplify the ordering issue as well as coherence maintenance, shar-
ing could be forbidden so that no duplication exists among L2 caches. The conse-
quence of this would be frequent data movements or cache line thrashing between
L2 caches. Given the built-in task deployment policy (c.f. section 2.3.2), recurrent
cache line exchanges could be restricted to adjacent L2 caches. Chip-wide
communications resulting from very irregular applications might be mitigated by
locality-aware coding.

Alternatively cache banks can replace these uniform caches in fig. 7.1 to con-
struct a Non-Uniform Cache Access (NUCA) memory [KBK02]. Since all banks
form a singe L2 cache, coherence requirement within the same cache level does
not exist any more. Then we could follow the simpler static NUCA strategy if
the latency of fixed cache access is respectable or leverage a bit more complex
dynamic NUCA to speed up accesses to frequently used data. The mapping,
searching and migration problems of the latter have already been studied exten-
sively [KBK02, KLI08, HFF09, MPG10, LMG11].

Stepping ahead from NUCA memory, we can get rid of any hardware coher-
ence maintenance by substituting scratchpad with private caches. Note, also a
combination of a banked shared L2 cache with scratchpad memories locally is
quite attractive. Applications can manipulate scratchpads directly to ensure data
consistency. Given our programming and memory model, there will be few ad-
ditional overheads to coding. Meanwhile this would bring a new direction to the
current scalable system research other than being entangled in the scalability issue
of on-chip hardware coherence.

Details and feasibility of above memory subsystems still have to be examined
and analyzed carefully before being put into practice.

7.2.2 Real-time stringency

The experimental results in fig. 6.4 and figs. 6.7 to 6.11 have disclosed the impact
of resource sharing, which is reflected as inconsistent execution time of equally
important threads in aperiodic RT, or execution jitter of the same RT when run-
ning periodically with normal tasks in the background. We have suggested an
unverified remedy for the former by dynamically adjusting priority levels elabo-
rated in section 6.2.3, and the solution for the latter as bandwidth amelioration
in section 6.3.2.6.

Dynamic priority may be propitious to use cases that are sensitive to the
deadline of an individual thread, e.g. wireless communication in packets, but not
to those that as well emphasize the execution time of the entire task. Investing
in bandwidth at the chip boundary is preferable to periodic use cases like safety
critical controlling, but it is less feasible for a limited hardware budget. Containing
concurrency from the software side offers a flexible alternative to bound deadlines
further given a fixed platform, but it is at the price of task performance and system
utilization rate as shown in figs. 6.14 to 6.15.

One step forward, when putting all periodic experiments together, it is not
difficult to find that the RT jitter is influenced by the accompanying background
task. This is to say timing results are always dependent due to the sharing of
resources, particularly the data path to memory. Consequently, provisioning independent or even predictable deadlines has to resort to privileged use of shared resources or even dedicated path to completely break the sharing bottleneck. This is the key for hard real-time processors and also a deficiency of the current priority scheme. For that purpose, the real-time privilege could sweep from cores to the chip boundary. A RT-biased private D-Cache is optional. If demanded, this can be made by partitioning current shared D-Cache or integrating a separate scratchpad. A new partition or scratchpad need not be of a large capacity, as our experiments show that the default 4KB D-Cache of each DRISC core in Micro Grid is sufficient to afford a sound pipeline efficiency, unless irregular real-time applications run in large parallel scale, e.g. 8 or more threads per core. Then a second bus could be equipped only for the use of RT to reduce inter-core contentions. This accordingly mandates the prioritized processing at the shared L2 cache side and at the memory controllers. With such a data path, real-time threads can achieve even tighter timing bounds and shorter execution time.

When scaling to a real-time processor with more cores, additional work has to be done at the system level on top of existing individual DRISC cores in the interest of both scalability and real-time stringency. Firstly a crossbar may be more appropriate than a bus; secondly a multi-banked L2 cache is much more desirable to allow concurrent accesses; thirdly more memory controllers are preferable. We believe all of the above methods could help not only achieve scalable performance but also approach more rigorous real-time requirements as well as higher processor utilization rate, and hence lower power dissipation for a given performance requirement.

Besides, we still have to investigate how to tailor on-chip resources to fit a given set of use cases and whether it is necessary to support more than two levels of priority in the hardware. Both are of course for the sake of a compact chip.

7.3 Other DRISC related research and issues

The research that has been explored in this thesis represents two kinds of development based on DRISC, namely general-purpose and specific systems, both of which aim at practical uses of the DRISC architecture. To achieve this still needs a lot of efforts. For example, apart from the performance scalability discussed in chapters 4 to 5, to always fit the power consumption into an envelop is also of great importance to a scalable chip with energy sustainability; the miniaturization of transistors makes cores error-prone in both manufacturing process and usage, and thus vulnerable to external interferences. This gives rise to the reliability issue, which is also frequently stressed in real-time computing and should be reflected in our real-time studies.

7.3.1 Reliability

There is on-going research based on the DRISC architecture to achieve reliability with fault tolerance. The research follows the classic redundant execution to capture errors and applies redundancy at the thread level across pairs of cores. More precisely, cores are paired statically; threads and their redundant copies execute on the paired cores separately. Memory changes from paired threads are checked
before being committed to the shared L2 cache. If there is a mismatch, the target thread has to roll back to the beginning of a customized region and starts a recovery.

Currently the first step, namely error detection is achieved. By leveraging locality within cache hierarchies and existing DRISC scheduling, [FYP+13, FYP+14] verify that enabling memory checking in instructions always gives rise to $<100\%$ overheads even in a many-core system with massive concurrency or only 11% in a single core mode. Nevertheless what is more challenging is the recovery.

Due to the DRISC programming and execution model, the finest granularity of recovery out of detected errors is inherently settled, basically an individual thread. Its availability depends on where the captured error happens. Additionally, flexibility is provisioned to define critical regions in programmes through language extensions. Only within these regions, error detection will be toggled and recovery will be made from the entrances. This is named on-demand redundancy and the coarser granularity replacing default behaviors in hardware could further reduce additional overheads. The study of this has been carried out at the thread level. It uses PC for checkpointing without introducing additional hardware. Preliminary results show a promising $<5\%$ overhead under the fault-free condition.

Re-execution is valid for soft errors or transient hard errors but not for eternal crashes like a permanent core failure. The redundancy strategy should be capable of catching such a kind of issue and report to the global resource manager which then masks off the defective cores. Given a homogeneous many-core system, fungible cores or resources are always available and isolating failed parts can have trivial impact on system performance. We are just working hard toward all of these targets.

### 7.3.2 Power awareness

Power consumption is one of the dominant factors in the evolution of architecture design and is of paramount importance. It used to enforce the shift to multiple cores but “dark silicon” in the coming many-core era will be more serious if not managed properly. This is why PPW is considered as an critical index to evaluate an architecture. During the exploration of the DRISC architecture, we repetitively emphasize average core efficiency as a way of showing energy efficiency; yet enabling power saving to always fit energy envelopes in scalability is rarely referred to.

As to well known techniques namely Dynamic Voltage and Frequency Scaling (DVFS) and thread migration for this purpose, the former is much more pragmatic to DRISC, but how to perform regulation is still open for discussion, e.g. the steps, granularity. Actually DRISC itself essentially gains advantages in power efficiency over conventional architectures. [BJ05] attributes this to the conservative instruction issue policies as

- no power is dissipated on speculative instruction fetch and execution;
- no area and power are required in making branch or data predictions;
- no area and power are required in managing miss prediction cleanup.

Thanks to the data-flow scheduling, more aggressive local power-control strategy can be exploited. Besides the scheme to shut down unused cores but only maintain
a set of powered up cores for immediate use to minimize static power dissipation, it is also possible to stop clocking on engaged cores but without active threads at the moment to further reduce dynamic power dissipation. In both cases, asynchronization interfaces are mandated to wake up (powering up or restarting clocking) sleeping cores directly or by signals upon incoming events, e.g. remote requests or the return of asynchronous data. Reactivated cores then either serve the incoming requests or schedule suspended threads to resume execution. The whole procedure can also be seamlessly integrated into existing scheduling mechanism and avoids data moving penalties due to thread migration.

### 7.3.3 Compatibility

From the very beginning, DRISC targets a compact and elegant architecture in contrast with “monolithic” conventional peers. To achieve this, DRISC follows featured mechanisms such as data-flow scheduling, the conception of micro threads grouped into families and concurrency management in hardware. The expected benefits can only be accomplished by using corresponding programming models with dedicated interfaces. All make DRISC unique from others but at the same time lead to compatibility issues. Leaving compatibility aside helps promote architecture design and innovation, but it in the mean time leads to the embarrassing situation with respect to putting the architecture into use, unless there is a well-built ecosystem of industry or wide cooperation within academia. This usually is the common challenge of academic research and innovation.

As a matter of fact, the peculiarities of DRISC have been gradually fading in recent years and this to some extent ameliorates incompatibility. [Pos13] summarizes these as

- exposing concurrency explicitly in programming is being introduced into emerging high-level languages, e.g. Scala, Haskell;
- managing concurrency in hardware also appears e.g. in Tilera’s TILE, and software frameworks like qthreads and Erlang’s run-time system are designed to manage bulk concurrency well;
- the relaxed memory consistency requirement has been exploited in some SIMD/SPMD accelerators, e.g. those leveraging the “gather-scatter-gather” streaming programming model.

Yet this cannot hide the weakness or lack in preemption support, proper exception handling, and the management of interruption and atomic operation (there are alternatives in DRISC), most of which are fundamental to existing OS and applications. Consequently it is impossible to piggyback on current operating systems or numerous benchmark suits directly without non-trivial refactoring.

Given unforeseeable challenges and difficulties of shaping the mature DRISC environment for general-purpose use, concessions have to be made to conquer incompatibility and for the practical usage of DRISC. A five-year research plan submitted to ASCI referred to an infrastructure study to achieve software implementation of the concurrency model on top of existing and emerging multi- or many-core architectures, for example the SCC. [Pos13] also suggests the following

- make specific implementation of the lacks to embrace popular software frameworks though optimal performance less attainable;
7.3. OTHER DRISC RELATED RESEARCH AND ISSUES

- tailor DRISC to cater for the need of the software community;
- extract features from DRISC and patch them to existing architectures that possess and show functional similarities on account of the accomplishment probability;
- strip DRISC down to an affiliated accelerator on a general-purpose chip to get somewhat more integrity.

An alternative path we are endeavoring to follow at the moment is aiming at specific applications and systems. This is just the object of the real-time study in chapter 6 and the reliability work in section 7.3.1.