On the exploration of the DRISC architecture

Yang, Q.

Citation for published version (APA):

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: https://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 2

Insight into the DRISC architecture
—exploring the abstract models

Abstract

This chapter emphasizes the fundamentals of the DRISC architecture. It discloses the motivations, ideas and principles interspersed among the programming and execution models.

Contents

2.1 Retrospect and history ................................................. 22
2.2 Programming model .................................................... 24
2.3 Execution model ....................................................... 30
Summary ................................................................. 35
2.1 Retrospect and history

The initial study of the DRISC architecture goes back to the uniprocessor days in the mid 1990s' when both multithreading and multiple issues were prevailing. The question raised in [BJM96] was given the efficiency of multithreading to hide latency and its rapid growth with the number of threads, is it possible to achieve the same result through a compact and neat RISC architecture in lieu of the conventional heavyweight multithreading solutions. It in turn addressed two issues: the more lightweight thread and more efficacious thread scheduling to cope with long latency and non-deterministic events especially in contrast to static schemes. This gave birth to DRISC with micro-threading and data-flow scheduling.

It is possible for a micro thread to have just a few instructions and several of them can be derived from within a single context and differentiated from each other by the unique PC. The imperative micro thread can be forked, joined and even synchronized with extremely low overheads. This is the solution to the problem of lightweight threads.

Dynamic scheduling piggybacks on the merit of data-flow machines where a scheduling decision is made only on the availability of required data\(^1\). This gets rid of the deficiency of static scheduling and facilitates synchronization in a sleep-wake manner.

Subsequent research [JL00, Jes04, BJ05, Jes06, BHJ06] concerning the DRISC architecture stressed its advantage in exploiting not only TLP but also ILP when used on CMP. This was especially compared with those leveraging aggressive technics e.g. wide issue with out of order or speculative execution, causing problems of hardware complexity and costs, pipeline utilization and power consumption. The CMP with DRISC cores then aimed at scalable concurrency and dealing with increasing asynchrony\(^2\) on chip. In consequence, the original ideas of DRISC were refreshed and further elaborated with how to program this architecture, how to perform communications and synchronization via registers and how register files are organized and implemented, how to sustain scalable thread management and how to improve dynamic scheduling with supports from compilers. All envisaged the embryo of a DRISC CMP with the design norm as “the area occupied on chip is proportional to the number of instructions issued per cycle (physical concurrency) also that the area occupied is proportional to the amount of latency tolerance required (virtual concurrency) and that both can be scaled over orders of magnitude [Jes06]”.

With existing fundamentals, studies of and around DRISC were boosted and accelerated by several consecutive projects and systematic technical collaboration within these projects in the following years, e.g. NWO Microgrids\(^3\), ÆTHER\(^4\), Apple-Core\(^5\) and ADVANCE\(^6\). On the core side, there is a clearer picture of the

---

\(^1\)Dynamic parallelism another feature of data-flow machines is also on the same basis but is abandoned here as it is not compatible with imperative programming.

\(^2\)e.g. inter-component communications because of the unproportionately scaled wire length and latency. How to overlap these asynchronous events with computations is vital to CMP performance.

\(^3\)http://csa.science.uva.nl/research/projects/microgrids

\(^4\)http://www.aether-ist.org/

\(^5\)http://www.apple-core.info/

\(^6\)http://www.project-advance.eu/
2.1. RETROSPECT AND HISTORY

CMP and it has steadily evolved to support more DRISC cores on chip as a many-core system. This is prompted with the abstract programming model to expose virtual concurrency and the concurrency interface backed with ISA primitives to bridge software concurrency and the hardware micro threads [Jes08b]. Meanwhile dynamic resource and concurrency management are processed by on-chip logic as an OS in silicon to take concrete jobs [Jes08a, JPvT08, van13]. These are later substantiated in the Micro Grid [JLZ09b, JHL*10, PLY*12, PLY*13a], the DRISC many-core system with scalability and some other features and functions beyond the suggested models. On the peripheral side and also the research goals of those projects, infrastructure construction for the many-core system has had desirable yields. The proposed programming model is achieved in the specific high-level imperative and sequential programming language via an incremental design on top of C. The matched compiler and libraries help expose it to different RISC ISAs [pos12a, PLY*12]. By virtue of international cooperation, the system is also a target in the design of the functional data-parallel language SAC and the asynchronous coordination language S-Net. With the help of these high-level programming languages, a myriad of applications can be reprogrammed or ported toward many DRISC cores and execute on the software simulator [LPY*13, PLY*13b], which adopts a component design with discrete events driving and cycle-accurate execution. With those auxiliary frameworks, it is easy and flexible to perform experiments and evaluation. The output of the SPARC DRISC core on FPGA is from project partners [DKK*12] which checks whether the architecture is accomplishable in hardware and what the cost will be, especially the unique requirements like proposed register file and thread management structures.

Figure 2.1: The landscape of the contemplated ecosystem. Normal arrow shows a “derivation” relationship while dashed arrow “supporting”.

Diagram showing the ecosystem landscape with software, proposed programming and execution model, abstract, high level language compilers, tool chain, specific, hardware, micro grid, simulator, FPGA, and Leon3.
```c
int main()
{
    thread T_dim(i,j,shared sum)//inner
    {
        index(k);
        get(sum) += A[i][k]*B[k][j];
        set(sum);
    }
    thread S_dim(i)//middle
    {
        index(j);
        create(N,BLOCK,...)T_dim(i,j,0);
        sync();
        C[i][j]=get(sum);
    }
    thread F_dim() //outer
    {
        index(i);
        create(N,BLOCK,...)S_dim(i);
        sync();
    }
    return 0;
}
```

Listing 2.1: Pseudo C code to express parallelism in programming via creating families of threads.

Figure 2.1 is the overview of the up-to-date achievements and their relationships inside the ecosystem. The remainder of this chapter puts emphasis on the upper left quadrant—the abstract models comprising the mature DRISC architecture as what it is and why it is made to be so. Some features specifically toward many-core purpose are also included for integrity. Such dissection will set out the various issues from programming models down to possible implementation considerations. While the exact hardware strategies are the major content of chapter 3 in part II which covers Micro Grid and the simulator too. The details of the software tool chain and the DRISC core in FPGA are left to their own literature, e.g. [Pos12b] and [DKK+12].

### 2.2 Programming model: the definitions

#### 2.2.1 Concurrency manifestation

The introductory example in listing 2.1 exhibits how the $N \times N$ matrix multiplication is converted from the three-level nesting for loops into the suggested fully paralleled version. This addresses the most critical part of parallel programming—how parallelism is exploited and expressed. It challenges the software development
2.2 PROGRAMMING MODEL

Figure 2.2: The concurrency tree according to exposed parallelism in listing 2.1.

Habits formed in the uniprocessor era when implicit parallelism is often assumed as a matter of course because of the hardware efforts in the background, which does not work any more these days in order to make full use of the computing powerhouse out of many cores. As a result, the inherent parallelism of applications should be exposed to hardware explicitly but programming languages achieve this in various ways.

As shown in listing 2.1, each loop level corresponds to a `create()` for a “tuft” of threads to run that level’s thread function. The potential parallelism among nesting loops is exploited accordingly by the nested creation of threads. The “tuft” is named a family; thus there are three families of threads structuring a hierarchical concurrency tree (fig. 2.2) that grows dynamically with family creations executed in the code. Each creation in the code will derive a sub-level of nodes as child threads (or sibling threads to each other). All nodes of the same hierarchy present the scale of parallelism on that level. Therefore, the closer to the root, the coarser the concurrency granularity; and the number of leaves indicates the maximal and finest level of virtual concurrency exploited and exposed in software.

It is clear this concurrency expression also adopts the fork-join pattern to express parallelism but it reflects the programming feature of uniformity, explicitness and independency.

2.2.2 Uniformity

The `create()` is the uniform interface to expose concurrency for both data- and task-level parallelism like POSIX threads (pthreads) [But97]. OpenMP [Ope13] and Cilk [Int10] have equivalent keywords and at the same time provide others to for loops. Although the shortcut may moderate the workload of refactoring existing code, it is not always convenient and is confusion given these restrictions. The parameterization of creation is another important part. It is drawn from
the meta data bound with the create() interface as the guidance of how to create these child threads. Rich information can be available there, for example all iterator attributes of a for loop determining the scale of the forked concurrency and the number of workers (not all appear in the pseudo code. Chapter 6 will use such information for the purpose of real-time computing). Again, this is unlike pthreads, OpenMP and Cilk that rely on a couple of environment variables that have to be set individually by separate imperatives before fork and thus are error-prone.

### 2.2.3 Explicitness

The discussed uniformity de facto addresses another feature, the explicitness as well, namely what you get is just what you see. The expression of parallelism is rather direct and highly comprehensible. The hierarchical tree shown in fig. 2.2 can be easily envisaged after reading the code without any further explanation in detail. Moreover, the interaction within a family of threads is clearly revealed and defined through the parameter list of each thread function.
Variables behaving as communication channels between threads are of two types: *global* and *shared* (these are also keywords for declaration). *Globals* are writable in the parent thread but are read-only to all child threads. *Shareds* are written-once in a thread and are only readable to its immediate successor in the thread index sequence. Note that parent thread has to initiate *shareds* for its first child thread and can get the final values of these *shareds* from the last child thread (fig. 2.3). Taking code in listing 2.1 as an example, thread $F_{\text{dim}}$ shares one global channel (the index $i$) with child $S_{\text{dim}}$ threads; thread $S_{\text{dim}}$ shares two global channels ($i$ and $j$) and one shared channel (sum) with its child $T_{\text{dim}}$ threads. Each thread $T_{\text{dim}}$ has to get the shared from its predecessor and set the shared for its successor as of communication. Thread $S_{\text{dim}}$ initializes the shared *sum* to zero for its first $T_{\text{dim}}$ child and gets the final result ($C[i][j]$) from its last $T_{\text{dim}}$ child. Note that the pseudo code considers matrices $A$, $B$ and $C$ have global visibility for simplicity; if not, they should be declared as *globals* in all thread functions’ parameter lists and passed from thread $F_{\text{dim}}$ down to thread $T_{\text{dim}}$. This is in contrast to the data sharing attributes of OpenMP which defines only data accessibility but nothing else.

As per communication types, families are sorted into two types: *independent* and *dependent*. Those with global channels only are independent families like the family of $F_{\text{dim}}$ or $S_{\text{dim}}$; those with shared channels are dependent families like the family of $T_{\overline{\text{dim}}}$ wherein a thread’s execution will be paused by the synchronization of *shareds* with its antecedent neighbor. The shared channel is an alternative to reduction of OpenMP and Cilk to eliminate data races, which creates local copies in each thread to keep concurrency intact and reduces all copies into the final shared value when gathering up all related threads. By comparison, the proposed *shared* communication is more rigorous in terms of accessing sequence and thus more deterministic. However at the same time it will impact the inter-thread concurrency as only those operations independent upon the synchronized *shareds* in a thread can proceed concurrently, otherwise be sequentialized by communication. Such overhead can be minimized by exploiting data locality and core affinity.

In a word, the communication pattern prescribes explicit and forward only data flows. This on one hand breaks any cyclic data dependency to ensure deadlock freedom and on the other hand imposes determinism to parallel execution expediting error tracing and debugging. Furthermore, it facilitates safe composition from tasks to functions and hints the run-time mapping to minimize communication costs. Finally it exposes concurrency to the compiler at the instruction level that can be exploited in high level constructs that are sequential in nature.

### 2.2.4 Independency

The character of independency is to say virtual concurrency in program is completely isolated from its implementation, either in software or hardware. Programming has no notion of these things or even has no need. This is a goal of our research, to break coding intertwined concurrency engineering to promote flexibility and scalability.

With it, the role of programming is only to exploit and expose potential concurrency and nothing else, and all the other things in the black box manage to
satisfy virtual concurrency as far as possible. Hence, the only similarity shared with OpenMP and Cilk is to declare the number of workers\textsuperscript{7}. Directives like grain size, chunk size or scheduling method to guide compilers are unnecessary, neither are postulated at the compiler side.

One specialty is that family execution is made concrete by binding each with some resources. The resource here is purely an abstract parameter indicating the individual requirements (e.g. the number of cores) and has nothing to do with the target chip. The following section will talk more about this.

Generally, the independency enables resource agnostic and parametric concurrency. This implies that once written and compiled, the binary acquires high adaptability and the performance is scalable assuming different execution platform without notorious labor-intensive refactoring and optimization.

2.2.5 Resource virtualization

We have alluded that even though DRISC architecture is the basis of the recommended many-core system, asymmetry and heterogeneity are desirable if demanded. The proposed parallel programming has already paved a way for various processing elements given the interface independency with virtualized resource reference.

On account of the diverse computations and the benefits from customized processors, desirable resources should be declared explicitly in the code in order to get better performance in addition to explicit concurrency. OpenCL \cite{Khr11} is the pioneer and exemplar of this, but it is also notoriously known for having complex interactions and coordinations among different types of processors in the programming interface. It is much more preferable to offer a simple and compact Application Programming Interface (API) to ease coding. As a step toward this, the concept of abstract resources for programming is introduced, the \textit{named place}.

A \textit{place} can be a cluster of DRISC cores for concurrent jobs, a monolithic CPU for sequential code and public services or other special processing units (e.g. GPU, Digital Signal Processor (DSP), FPGA or ASIC). Concerning programming, it only has to announce a measure of quality and quantity, and request the system resource manager for assignment. For example, from a certain processor pool the manager will allocate the requested resources and return the handler, “Place”, as the mnemonics shown in line 22 of listing 2.1 referred in create(). Allocated resources can be shared by different families using the same handler or the following names for convenience

- \textit{local}—axiomatically, “local” means the forked family of threads are about to run on a single unit of the place inherited from the creating parent thread, in a time sharing mode.
- \textit{default}—unlike \textit{local} which constrains the child thread to one unit of the inherited place, here the child threads share all resources with the family to which the parent thread belongs. If the parent family is constrained to a single unit, \textit{default} is equal to \textit{local}; otherwise the child threads will spread over all the processing units that support the parent family of threads.

\textsuperscript{7}“Worker” is conceptually identical but varies in implementation.
2.2. PROGRAMMING MODEL

for (i=0; i<n; i++)
    sum+=i;

(a) order insensitive

#core 1 #core 2
if (a==0) if (b==1)
    b=1; a+=2*b;
else else
    b=a+2; a=b+2;

(b) order sensitive

for (i=0; i<n; i++)
    printf("hello, this is %d", i);

(c) public service

Listing 2.2: Synchronization example in concurrent execution.

- exclusive—Each exclusive place is a special unit that only offers sequential execution. All family creations referring to the same exclusive place are forced to be queued at the target unit and served to run serially in the First Come First Served (FCFS) way. The specification of exclusive is instrumental in generalizing the execution pattern and offers an alternative to cope with arbitrary synchronization.

2.2.6 Synchronization

The shared memory programming model facilitates data exchange between parallel entities simply using load/store instructions without explicit messaging, but it has to address the synchronization issue, the common flaw of parallelism because of non-deterministic memory snapshot.

Listing 2.2 gives three examples that require synchronization for correct results but differ from each other. The typical data race in the first for loop is not susceptible to the access order and can be removed by setting sum as a reducer of OpenMP and Cilk. In contrast, code segment from line 11 to line 14 will yield different result depending on the relative execution speed of two different cores. Such nondeterminism can be eliminated by inserting barriers to enforce memory consistency so that each read could get the latest written value. Similarly, the join action gathers all threads within a concurrent domain where barriers are mandated to reach a definite view of the shared memory for the sake of subsequent correct execution. Logically there will not be any error in outputting a string on screen as
the last loop does but concurrent printing will break the integrity of such a sentence to produce messy strings like “hhelhelo. . . ”. A pragmatic solution is making the shared printf service as a critical section guarded by mutex (semaphores or locks) in order that only one thread can access it each time.

Synchronization not only poses difficulties with regard to concurrent programming and hardware implementation, but also introduces extra penalties to parallelism by pausing the current execution or imposing locked steps (e.g. handshakes among cores). Since most rely on barriers (at least in hardware) to function properly, prior work has sought for either lower cost strategies, for example [VPCCCR06] to ignore redundant memory synchronization, [SK10] and [AFA12] for cheap chip-wide synchronization, or alternatives like transactional memory [HLR10] to atomize memory operations, as found in the BlueGene/Q processor from IBM and the latest Haswell processor of Intel [Int12].

The explicit and forward only data communication in section 2.2.3 provides an effective alternative to guarantee data race freedom whereas the paradigm only pertains to intra-family interactions. There is no such assurance for inter-family communications presuming accesses to shared memory blocks. Exclusive places then come into this scenario because of the sequential execution nature. Similarly, it is convenient to substitute the guard from semaphores or locks for these critical sections due to the side effects (e.g. deadlocks). As a result, exclusive places function like “secretary” of [Dij71]. This and a typical use case have already been explored in [PGA+12]. Readers can move there for details.

To minimize synchronization impacts, consistent memory is only limited to the parent thread and its children, and is only promised before entering into and after leaving the concurrent region. Specifically in listing 2.1, all writes to memory in main() before create() in line 22 must be seen by all \(F_{_dim}\) threads no matter which place they use and all memory changes by \(F_{_dim}\) threads must be visible to subsequent code after the line 23 sync(). This implies mandated write barriers at these two points. Any other scenario asking for consistent memory can resort to additional barriers or exclusive places.

2.3 Execution model: run-time details

2.3.1 Concurrency in ISA primitives

Explicit concurrency as families of threads from the programming model does not consider the conventional way in which these are mapped to worker threads and scheduled by OS kernels with the help of run-time libraries; alternatively it can be exposed to hardware directly after being compiled into instructions akin to Transputer [HMSS87, dM92] that is able to instruct and manipulate processes through special instructions like STARTP, STOPP, RUNP and even the microcoded scheduler to replace the OS kernel’s role.

Controlling concurrency at the instruction set level follows an incremental design on top of existing RISC ISAs with minimal additional primitives like:

- \textit{crt}—Such instruction initiates bulk thread creation at the target place. All cores in that place perform asynchronously to finish creating threads and then automatically start their execution. Information about place handler
and thread entrance PC has to be included or indicated in its operands. A returned family handler is required to feed back the creation results and for later use;

- **sync**—Such instruction requires the creating thread to stop until all created threads complete. Usually, it imposes bulk synchronization to the target family of threads and the thread issuing this instruction (host thread). The host thread has to synchronize with the completion of all target threads. The target family handler is mandated in its operand. A return value is optional to record the synchronization result;

- **brk**—Such instruction is similar to the break of loops in that it stops creating more threads. It is required in problems without definite parallel scale, e.g. numerical approximation. When certain condition is met in one thread, un-created threads are then superfluous and can be canceled to save time and energy. *brk* should affect the host thread’s family only and thus has no need to take any operand. It relies on virtualization for any benefit as it is a preemption on creation not on execution of threads;

- **kill**—As its name implies, such instruction terminates the target families of threads dropping all results and collecting all occupied resources. On account of its impacts, *kill* should be under privileged use. It may be helpful to global management or fault recovery. Also, the target family handler has to be appended to its operand;

to undertake the management in hardware, yet additional services are still required for completeness.

### 2.3.2 Task distribution

Thread creation first pledges load balancing, namely software threads are evenly distributed inside the place. Assuming the family of \(F_{\dim}\) threads in fig. 2.2 bound with a place of \(C\) cores, each core will receive an assignment of \(N/C\) threads in the way that thread 0 ~ thread \(N/C - 1\) run on the first core, thread \(N/C\) ~ thread \(2N/C - 1\) run on the second core and likewise according to their logical index. Such a block distribution takes locality into account and tries to maximize shared data overlapping amongst sibling threads, though there may be optimal data partitions for irregular applications.

The minimum of \(N/C\), desirable number of workers (BLOCK in listing 2.1) and the supported hardware threads decides the maximum concurrency of \(F_{\dim}\) family on a core, assumed to be \(M\). \(N/C\) software threads by and large iterate through \(M\) hardware threads serially to execute the same thread function. Thus \(M\) empty slots and associated resources are sufficient to support the \(F_{\dim}\) family on a core presuming that \(N/C\) software threads enjoy resource sharing and reusing.

It may be possible to assume the targeted place comprises idle cores to meet the families resource requirements. However, it may also be that the resources are shared between different families. Supposing no other place is available and all families in listing 2.1 have to share the same place conforming to the same distribution, then the resource pressure will increase as the cube of the problem size \(N\) and will soon be beyond the capability of an individual core. Considering the parallel computation starting from top to bottom with creations and finishing from
bottom to top with synchronizations along the hierarchical tree in fig. 2.2, deadlock would occur in the top-bottom procedure as soon as any resource depletion appears. This is a major challenge to dynamic resource management.

2.3.3 Resource management

Software concurrency is virtual and cannot be guaranteed in hardware because the mapping is not straightforward and is vulnerable to resources. As to our case, this is subject to the appropriate cores to form a place and the available hardware resources of each core.

For either create() or crt, a place handler is compulsory which is returned by the global resource manager according to resource announcements in the program. With regard to programming independency, it is quite probable that the target system is disqualified in either the property of cores or the amount. Therefore, the creation of threads will fail or even the entire program will be unable to run at all. Thanks to the parametric interface, this embarrassment can be dissolved by lowering down the requirement to an affordable level without touching the binary code. However, resource allocation in creating threads is another matter.

Cores enabling HMT prepare contexts for each hardware thread. Related resources can be partitioned either statistically and evenly for every thread or dynamically shared among all threads. The former ensures a ready worker to execute instructions at any time unless all affordable hardware threads have already been occupied. This may lead to inefficient resource usage as well. On-demand allocation from the shared resource pool will perfectly fit an individual requirement but may cause starvation with abusing or greedy usage given the fixed total amount. The hybrid policy may be a good choice namely applying a threshold to each dynamic allocation.

No matter how resources are allocated, it is never possible to circumvent the problem of no available resources or free thread slots when a thread creation happens. Three candidate solutions are available

1. Cancel this create. The host thread can try again at other time, change to another available place (this requires changes to the source code) or stop the program and exit.
2. Suspend this create. The creation request will be queued at the target core and wait until sufficient resources or workers are attainable again. This is preferable if the host thread need not synchronize with the forked family, e.g. call printf() to output a string, or else it will suffer variable latency.
3. Compromise on this create. A compromise is plausible merely when there are some resources but not enough. Then virtual concurrency scales to smaller scale hardware parallelism (the number of ready slots) down to the sequential execution (only one ready slot is available or sharing resources with its parent and substituting a normal function call with a create).

Once with acquired resources, a hardware thread can enter into pipeline to start execution. It may be switched away by the predefined routine of the run-time scheduler but unexpected latency such as suspended creation should be handled properly to improve system performance.
2.3.4 Dynamic scheduling

Section 1.2 has introduced different multithreading strategies. Thread interleaving at every cycle impinges on both instruction and data feeding speed demanding either large hierarchical caches or high bandwidth memories. This applies as well to multiple issues but more hardware, expensive speculation and the ensuing costly recovery on speculative failure. Therefore, the DRISC pipeline aims for simplicity but competitive or even better thread scheduling to hide latency, reduce stalls and improve system throughput, which mainly resorts to data blocking.

Thread scheduling can borrow from dataflow machines such that all execution is only driven by data availability. A thread will be switched out of pipeline when its instruction misses operands and switched in when they are ready. The switching decision is made at the point of data use in contrast with at the long latency instruction itself that leads to such missing data. For example, an ordinary control flow machine regards a load miss as the critical point to initiate either thread switch or speculative execution, but here the same thread’s instruction stream continues flowing in pipeline until one instruction misses its source operands at Arithmetic Logic Unit (ALU) input ports. As a consequence, such a dataflow strategy not only covers what conventional static or dynamic blocking does, e.g. compiler tags, cache misses or interruptions, but also captures most non-deterministic latency (e.g. suspended creation) to yield more efficient multithreading. Moreover, this paradigm permits cheap barrier synchronization in both implementation and execution. A barrier just enforces waiting on a certain value; thus communications can be overlapped by computations through thread switching and data checking can be triggered by any change to the value spontaneously even without spin locks. Meanwhile the concept of “data” could be further extended to cover anything that hold up the smooth running of a thread. Latency therefore might be hidden by dataflow scheduling if appropriate. Instruction fetch miss is an exemplar of this, as it is possible to switch out a thread when it can not be guaranteed to hit the instruction cache (I-Cache) with its PC and reschedule it when the correct cache line is present.

Context switch is quick and cheap here with the penalty of 1 cycle to the thread switched out (i.e. the reissue of the same instruction when missing data becomes available). This may not be the complete picture because of pipeline stage flushes. Assuming enough concurrency and a switch decision at the execution stage, at least two stall cycles (flushing fetch and decode stages) will be introduced before another thread enters into pipeline. To conquer this, a compiler optimization helps by either inserting switch tags into the binary at compile time to hint that an operand may not be available, then a switch may occur at instruction fetch stage without any flush (another ready thread can switch in as soon as the following cycle bringing no bubbles); or prolonging the dependency distance as does for ILP to reduce the switch frequency.

Efficient scheduling would keep a busy pipeline and as well produces the thread life span shown in fig. 2.4. In addition to keeping the updates of PC in every pipeline switch, extra information is mandated in the procedure to maintain concurrent entities.
Figure 2.4: States transition of a thread in its lifetime.

<table>
<thead>
<tr>
<th>id</th>
<th>work to do</th>
<th>allocated threads</th>
<th>peer families</th>
<th>.....</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5: Required information to support a family or thread and the suggested organization.

<table>
<thead>
<tr>
<th>id</th>
<th>PC</th>
<th>sibling threads</th>
<th>parent</th>
<th>.....</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>PC</th>
<th>sibling threads</th>
<th>parent</th>
<th>.....</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.3.5 Concurrency maintenance

It is not rare for different families of threads to share the same core at the same time. This raises problems of how to differentiate all these threads, and how to reflect the ownership and affiliation of them. These two issues are crucial to an individual family or thread throughout its life.

Section 2.3.2 demonstrates the task distribution and iterative execution pattern; thus at the family level, untouched workload and already running child threads must be recorded for the purpose of reusing resources. The same is true for peer families on other cores in case of any communication required (e.g. synchronization). As to a certain thread, it has to remember which family it belongs to so as to return occupied resources to those waiting software threads of the same family at retirement; it also has to remember its immediate neighbor for possible data communications and synchronization. Storing the PC is also essential to keep the interrupted point from where it later resumes execution.

Putting these together, each family or thread should have a unique identifier (within a core and other than its logical index) bound with which is the desirable information. This indicates a piece of organized memory to maintain all such information. An example is shown in fig. 2.5. Given the rather frequent accesses (thread creation, switching and retirement within the same cycle), on-chip storage is much more preferable to off-chip memory, which in turn becomes the object of resource management and a factor restricting attainable concurrency.

Summary

Fully hardware independent parallel programming is attractive and our efforts toward this target were introduced and compared with other state-of-the-art products. Programming convenience exacts sophisticated compiling techniques supported by operating system and/or hardware. Accordingly the innovations of the proposed DRISC architecture were investigated mainly around its direct backing of virtual concurrency. Most of these approaches are abstracted and are just prototypes, and how they are achieved is up to the specific implementation.