On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges
Poss, R.C.

Citation for published version (APA):
Chapter 3

Architecture overview

Abstract

Here we provide an overview of the architectural concepts that characterize hardware microthreading, and outline the architecture design principles which guide its implementation in hardware. We also present the implementation choices that have been made prior to our work.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>44</td>
</tr>
<tr>
<td>3.2 Core micro-architecture</td>
<td>44</td>
</tr>
<tr>
<td>3.3 Concurrency management within cores</td>
<td>48</td>
</tr>
<tr>
<td>3.4 Multi-core architecture</td>
<td>53</td>
</tr>
<tr>
<td>Summary</td>
<td>57</td>
</tr>
</tbody>
</table>
3.1 Introduction

In the previous chapter we have outlined an innovation space for CMPs with smaller, more numerous general-purpose cores, a distributed structure, focus on inter-core protocols, and HMT to tolerate diverse on-chip latencies. The work on hardware microthreading at the University of Amsterdam, performed mostly prior and outside of the scope of our own work, can be describe as a design activity towards new CMP structures in this innovation space.

The outcome of these activities is an overall CMP architecture model with custom on-chip components. Yet previous reports on this activity (in [BHJ06a, Has06, BHJ06b, Lan07, BGJL08, JlZ09a, JLZ09b, Lan1x] and other academic publications from the same authors) have focused mainly on general motivations and the eventual applicability of the proposed architecture, without providing comprehensive descriptions of the architecture itself in a way amenable to study by outsiders to the field, in particular software communities. The reason for this is that the academic communities around computer architecture are traditionally competing via performance results, leaving little publication space for describing components and design trade-offs.

To increase the visibility of the proposed innovation to external audiences, and provide a technical context to the remainder of our dissertation, the present chapter describes the general principles of hardware microthreading and its possible implementations. Together with chapter 4 and the accompanying Appendices A to E, this contemplative description constitutes our first contribution. This chapter is intended to complement [Lan07, Lan1x], which detail the design of the hardware implementation which we used as our main reference platform.

3.2 Core micro-architecture

The micro-architecture of individual cores favors simplicity, i.e. fewer gates per core to increase the core count on chip and the performance per watt, over sequential performance per core as suggested in section 2.2. It also combines dynamically scheduled HMT for latency tolerance with a short pipeline to tolerate branches, as suggested in section 2.6.

3.2.1 Multithreading and dataflow scheduling

- The design proposes to extend an in-order, single-issue RISC pipeline as depicted in fig. 3.1:
  - the fetch stage is extended to use PCs from a thread active queue (FIFO), provided by a thread scheduler in a Thread Management Unit (TMU). Control bits in the instruction stream read from the L1 I-Cache (“L1I” in the diagram) indicate when to switch to the next PC in the queue. A switch occurs also whenever the fetch unit starts to read from a different line in the I-Cache in order to ensure that no executing PC misses. If no PC is available, the fetch stage becomes idle;
  - the decode stage is extended to translate the register names listed in the instruction codes by a register window offset provided by the TMU for the current thread. This provides dynamic addressing in the Integer Register File (IRF) for different threads, as well as dynamic sizing of the register window in each thread;
  - the Register File (RF) is extended with dataflow state bits on each register, which indicate whether a register is full (has a value), empty (no value) or waiting/suspended (a value is expected, a thread is waiting on a value). Bypasses from the Arithmetic
Logic Unit (ALU) and Load-Store Unit (LSU) back to the read stage are also extended to carry the dataflow state of updated registers;

- the read stage is extended to check the dataflow state bits, and suspend the thread when a register operand is not full;
- the Memory Control Unit (MCU), actually part of the LSU but separated in the diagram for clarity, is extended to set the target register to the waiting state upon L1 read misses, but without stalling the pipeline nor suspending the thread as the next instructions may be independent; meanwhile, load completions are written asynchronously to the register file and wake up threads, as discussed below and in section 3.2.2;
- the writeback stage is extended to also propagate control signals from the thread management instructions (discussed further in chapter 4) to the TMU, and reschedule the PC of the current thread on the active queue if it was descheduled at the fetch stage but has not suspended in the pipeline;
- the newly introduced Thread Management Unit maintains state information for threads (PC, register window offsets, etc.) in dedicated structures on the core.

The notion of “dataflow scheduling” comes from the fact that threads are suspended upon reading from empty operands, and resume execution only once the operand becomes available. This asynchrony is implemented by storing the head and tail of a list of suspended threads in the input register when it is not full. When an asynchronous operation completes (e.g. a memory load), the register is updated by the corresponding unit and simultaneously the list of suspended thread(s) is appended to the active queue. Each thread context is described by an entry in a dedicated memory, called the thread table, which contains its PC and next field for schedule and suspend queues. An example is given in fig. 3.2, which illustrates the thread table after threads 5, 2, 3, 1 have tried to read from a waiting synchronizer.

\footnote{That is, the thread does not suspend on the load itself and independent subsequent instructions are allowed to execute. If some subsequent instruction is data dependent on the missed load, that instruction will suspend the thread if its operands are not ready yet.}
CHAPTER 3. ARCHITECTURE OVERVIEW

Side note 3.1: Active, ready and waiting queues.

For the purpose of a high-level overview it is sufficient to consider a single queue for schedulable threads. However, a naive implementation based on a single queue would be ill-equipped to deal with I-cache misses. Instead, the detailed design proposes to place schedulable threads on a ready queue, and only move threads from the ready queue to the active queue (connected to the fetch unit) once their instruction data is in the I-cache. Upon I-cache read misses, ready threads are placed in a waiting queue instead, and moved to the active queue upon completion of the I-cache read. See also side note 3.3, [Lan07, Sect. 4.3] and [Lan1x, Sect. Thread Scheduler].

The scheduling strategy is not purely dataflow oriented however. To avoid pipeline bubbles, the code is statically instrumented by control bits, visible at the fetch stage, which indicate whether an incoming instruction may suspend. If so, and there are more threads in the active queue, the current thread is popped from the active queue (descheduled) after the current instruction is fetched, and the fetch unit switches preemptively to the next thread’s PC for the next cycle. This effectively achieves zero-cycle thread switching.

This overall core structure requires only two read ports to the IRF for the read stage, one read/write port for the writeback stage and one arbitrated read/write port for asynchronous completions.

3.2.2 Long latency operations and completions

- As mentioned previously, upon L1 read misses the MCU writes back memory load completions asynchronously to the register file via a dedicated port. To determine which registers
to update upon completion, the LSU stores the address of the register to update in the L1 cache line\(^2\).

Dataflow scheduling can also be used for multi-cycle local operations. For example, integer multiply and divide can be implemented as asynchronous integer Functional Units (FUs) which write back their result asynchronously to the register file (fig. 3.3a), waking up any suspended thread(s) during the write if any. The same principle can be used again for an FPU, as depicted in fig. 3.3b. The Floating-point Register File (FRF) is equipped with dataflow state bits as well, and Floating Point (FP) registers are used in the same way as integer registers for the purpose of scheduling. Since the FUs are thereby fully asynchronous, it becomes possible to share an FU between multiple cores; this opportunity is exploited in the configuration studied in chapter 13. Note that despite the scalar structure, there is little contention on the asynchronous R/W port of the register file because at most one instruction is issued per cycle, and thus there is at most one completion per cycle on average [HBJ07].

In all these scenarios, the continuation of the long-latency operation, represented by a list of threads waiting on the value, is stored in the target register until the operation completes. To avoid losing this continuation and causing deadlock, any further instruction with the same register as its output operand will suspend as well or stall the pipeline until the first long latency operation completes\(^3\).

### 3.2.3 Threading to handle pipeline hazards

- A perhaps striking feature of the proposed design is the absence of several components found in other processors to handle pipeline hazards.

  As seen above, the design self-synchronizes true dependencies. Out-of-order completions from asynchronous FUs do not need reorder buffers, because further dependent instructions (both true dependent and output dependent) will self-synchronize in instruction stream order. A branch predictor becomes unnecessary once tight loops are replaced by dependent threads, one per loop iteration, interleaved in the pipeline. Branches are also marked to cause a switch upon fetch, so that instructions from other threads can fill the pipeline slots immediately following a branch while it is resolved.

  There are possible structural hazards, each with candidate solutions. A store to an L1 line waiting on a load may stall to preserve the ordering of memory updates. This can be solved by storing the continuation of the load in a separate structure than the line itself and merging stores with the incoming line upon completion. An instruction issued to a busy non-pipelined long-latency FU, such as FP divide, may stall if there is no space left in its input buffer(s). The alternative is to increase the number of FUs. A memory operation reaching the MCU from the pipeline at the same time as a memory completion may stall until the completion is effected. The solution is to dual port the cache and arbitrate updates at the granularity of single lines. In any case, solutions exist to avoid wasting the pipeline slot and rescheduling the thread. While these design choices fall outside the scope of our work, we mention them here for clarity and completeness.

---

\(^2\)Specifically, the L1 line contains the address of the head of a linked list of registers to update, since multiple loads can be waiting on the same line.

\(^3\)Whether this situation causes a stall or a suspension is still a topic for research, but is outside the scope of our work. In either case the architecture should ensure that continuations are not lost. In the early implementations, this guarantee was not provided, but this was later fixed, cf. section 4.6.1.3.
3.2.4 Critical latencies, IPC and throughput

- The proposed design issues at most one instruction per cycle; the throughput is determined by frequency, pipeline utilization and the number of dataflow misses, i.e. reads from empty operands.

  The two structures with non-trivial access latencies in the critical path of the pipeline cycle time are the register file and the L1 D-cache. While the number of ports on the register file is low, the number of registers increases with the desired number of simultaneously allocated threads. The size of the L1 D-cache, and thus its access latency, increases as the desired heterogeneity of workloads increases.

  To quantify the impact of these structures and predict the maximum allowable core frequency, CACTI [WJ96] estimates using conservative parameters have been computed. For example, given a 64-bit core implementation configured with 1024 registers per RF, a 4-way associative 4KB L1 D-cache, and support for up to 256 threads, the maximum allowable pipeline frequency at 65nm CMOS lies in the range 800MHz to 1.5GHz; we used these parameters during our evaluation activities, with a 1GHz pipeline frequency. At 45nm CMOS, the pipeline frequency can be increased to 2GHz.

3.3 Concurrency management within cores

The design provides dedicated hardware circuits to accelerate the organization of software concurrency during execution, as suggested in section 2.7. We explain this below.

3.3.1 Thread management

- In most processors, the program counter is initialized at hardware start-up to a predefined value, and from then on updated either by branches, traps or interrupts. The traditional vision is characterized by the assumption that the processor is dedicated to an instruction stream as long as it is powered. The logical activation and de-activation are indistinguishable from the physical events “start of world” and “end of world,” which cannot be controlled in software. This abstraction has been carried over to recent multithreaded architectures, for example Niagara and HyperThreaded processors, where each thread of execution is activated upon core initialization and subject to traps and interrupts in the same fashion as a sequential processor.

  In contrast to this, microthreading reifies “thread creation” (processor start up) and “thread termination” (processor shut down) as hardware events that can be programmed. The initial configuration of the PC, which in a traditional architecture is fixed statically at design, is also configurable dynamically during creation. However, the choice of which hardware resources (registers, PC slot) to use is not directly programmable; they are allocated dynamically by the TMU upon receiving a thread creation event. As we describe below, the arguments of the event determine the initial PC, as well as how many registers to allocate.

  This allows us to compare a conventional approach to hardware multithreading (fig. 3.4) with the microthreaded approach (fig. 3.5). In the conventional case, the thread is ready to execute instructions from a pre-configured PC as soon as the chip is powered. The thread may suspend upon blocking operations, or upon executing the HALT instruction which stops processing entirely. When halted, only an interrupt can wake the thread up and make it usable again. With microthreading, thread contexts are initially “empty,” meaning unused and not mapped to any workload. Upon receiving a thread creation event the TMU allocates
3.3. CONCURRENCY MANAGEMENT WITHIN CORES

Figure 3.4: Thread states in the Niagara architecture.

Figure 3.5: Thread states in the microthreaded architecture (simplified).

See also the detailed diagram in fig. 3.6.

Side note 3.2: New instruction vs. control bits for thread termination.

In an alternative implementation, a dedicated “terminate thread” instruction could be added to the ISA. However, since the architecture already uses out-of-band control bits for scheduling (section 4.4), these can be extended to provide control over thread termination without the need for a new instruction. This choice can also increase pipeline utilization for small threads with only a few instructions each: control bits do not require occupation of a pipeline slot, and threads can terminate and be cleaned up as a side-effect of their last useful instruction.

a thread context from a thread table, populates the context with a PC and makes it “active.” From that point forward the thread behaves like a sequential processor. The additional extension is that program-specified, out-of-band control bits in the instruction stream can trigger the thread termination event, which stops processing instructions from that thread and releases the context for another thread creation (cf. side note 3.2).

This basic thread management scheme implements the dynamic creation of logical threads over physical thread contexts, analogous respectively to tasks and worker threads in software concurrency managers. The resulting support for a dynamically variable number of hardware threads is a distinguishing feature of the design.

3.3.2 Preemption and asynchronous events

Traps and interrupts have been designed in sequential processors essentially as a means to multiplex access to the single-threaded processor between a program and an asynchronous event handler. When the processor has as many thread contexts as there are event channels, each event handler can run in its dedicated context, and support for control flow interrupts is not required in principle.

The microthreaded core exploits this opportunity by replacing support for external interrupts with support for a large number of thread contexts. For example, the reference implementation we use in our work supports 256 thread contexts. The reception of external events can then be implemented on a control NoC either by active messages [vECGS92],...
Figure 3.6: Thread states in the microthreaded architecture.

(a) Interface for active messages on a control Network-on-Chip (NoC). A Network Control Unit (NCU) disassembles incoming NoC packets to TMU control signals and vice-versa.

(b) I/O interface to translate generic I/O events to dataflow events and thread creation requests. The mapping between I/O channels and control signals is programmable via configuration registers.

Figure 3.7: Support for incoming external asynchronous events.
Side note 3.3: Fine-grained thread states.

The actual thread states are detailed in fig. 3.6:

- the interface to memory via an I-Cache suggests a discrimination between “ready,” “active” and “waiting” states, so the latter accounts for ready threads waiting to refill the I-cache;
- the First-In, First-Out (FIFO) nature of the active queue requires the pipeline to remove active threads from the queue upon switch, and add them back at the writeback stage if they are still schedulable. This in turn requires a distinction between “active” (still in FIFO) and “running” (not in FIFO, but still in pipeline);
- to prevent a “waiting” thread from never becoming active, I-cache lines must be locked until the corresponding thread gets a chance to run some instructions. To achieve this, the following conditions are met: the I-cache is fully associative; there are more I-cache lines than pipeline stages; and each line contains a reference counter with the number of waiting/active/running threads currently using the line, so that it can be evicted only once all associated threads exit the pipeline.

In hardware, these states are not named as state bits in a memory; they can be distinguished by which hardware components currently refer to the thread entry. Performance counters can be implemented that keep track of how many threads are in each state.

which carry an entire request for thread creation (PC, optional value argument) directly to the TMU, or by an I/O interface, which translates interrupt-style control signals to either dataflow events in the pipeline or thread creation events in the TMU. Both schemes are illustrated in fig. 3.7. Research is ongoing to implement scheduling priorities by placing threads created from asynchronous events in different schedule queues.

This preference for active messages and thread creations to signal asynchronous events over control flow preemption via interrupts is another distinguishing feature of the design.

3.3.3 Configurable register windows

- In a traditional RISC, register operands in machine code index the register file physically. There have been three extensions to this in previous work. In the SPARC architecture, sliding register windows are implemented by updating a Current Window Pointer (CWP) at function call and return. The CWP in turn offsets the instruction-specified register name to index a register file containing multiple 16-register blocks [GAB+88, MAG+88]. This structure was intended to optimize the overhead of function calls by avoiding spills, and in turn enables an optimization of the physical layout, due to the fact that only one window is used at a time [TJS95]. In superscalar designs, register renaming [Sim00] changes register names in instructions to index a larger physical register file than the logical numbering allowed by the ISA. In hardware multithreaded processors, each thread is configured to use a private thread register window in a separate region of the physical register file.

These advances have introduced a distinction between the register file, which is the physical data store, and the logical register window, which is the virtual set of register names, i.e. virtual register addresses, that can be used by machine instructions. The translation of names to a physical address in the store is performed in the issue stages of the pipeline.

The microthreaded core exploits this opportunity further and introduces two innovations: the ability to configure the size of the logical register window and the ability to configure the overlap between register windows of separate threads.

Variable window sizes are motivated by the need to increase utilization of the register file, which is typically the most expensive resource on the core. The argument is that tolerating on-chip latencies requires concurrent workloads that are short, possibly a few instructions. This in turn entails that some threads only require a few registers, i.e. potentially many
less than the maximum window size allowable by the ISA. By mapping only the part of the logical register window that is effectively used by a short-lived thread, it becomes possible to create more threads on the same number of physical registers and RF utilization is increased.

For example, the reference configuration we use provisions 1024 physical registers and 256 thread contexts. This leaves room for 33 threads with a fully mapped 31-register ISA window, or 256 threads with only 4 registers mapped in each, or heterogeneous combinations of any intermediate configuration. A dedicated hardware Register Allocation Unit (RAU) in the TMU performs the dynamic allocation and de-allocation of physical registers upon thread creation and termination. Upon exhaustion, context allocation is delayed until there are enough registers available to satisfy a request.

Meanwhile, overlapping register windows enable fast thread-to-thread dataflow communication and synchronization. When two instructions in separate threads target the same physical register, the dataflow scheduler (cf. section 3.2.1) ensures that the consumer instruction is scheduled only when the producer instruction completes. Moreover, if the instructions of both threads interleave in the pipeline, the bypass bus ensures that the data flows from one thread to another without any waiting time. While this concept seems challenging to exploit in a compiler or hand-crafted program, we shall explore possible uses in chapters 4, 6 and 8 and section 13.8.

This fine-grained register window management combined with the exploitation of shared registers for zero-cycle dataflow synchronization are yet two other distinguishing features.

3.3.4 Synchronization on termination

- Synchronization on termination, also called “waiting” or “joining,” is a fundamental feature of concurrency management systems.

  A common approach to implement this primitive is found in Unix, described for example in [MBKQ96, Chap. 4]: when a task terminates, it is moved from the “active” state to the “zombie” state, and its parent (if any) is notified with a signal. Independently, another task can use the “wait” primitive which suspends the waiting task until the target task terminates. When the target task terminates, the “wait” primitive completes and returns the termination status of the target task to the waiting task.

  The proposed architecture design implements this synchronization in hardware. A control event to the TMU, called request to synchronize can associate a thread context to the address of a register. When the thread terminates, the TMU writes a value into this register. Independently, another thread which has this register mapped into its logical register window can initially set the register to the “empty” state, and then start reading from it, which causes the thread to suspend. The waiting thread only resumes execution once the register is written to, i.e. when the target thread terminates.

3.3.5 Bulk creation and synchronization

- The features described so far are sufficiently general to support nested fork-join concurrency in programs, with threads created and synchronized one at a time. Yet the proposed design makes another step towards reducing management overheads.

  Considering SIMD/SPMD workloads of a few instructions over a large data set, the overhead of explicit instructions to create and synchronize every instance of the parallel workload would not provide any benefit compared to a partially unrolled sequential loop. Instead, the TMU provides a dedicated, autonomous Thread Creation Unit (TCU) in hardware which
3.4. MULTI-CORE ARCHITECTURE

is able to create and start threads asynchronously, at a rate of one every pipeline cycle or every other cycle, initialized by a single bulk creation event. A bulk creation event specifies a common initial PC for all created threads and a common register window configuration, including a common overlap factor between all created windows. To distinguish the created threads, the bulk creation event also specifies a configurable range of logical thread indexes assigned sequentially by the TCU to each created thread, and pre-populated in a private register upon thread initialization.

For synchronization, the TMU binds a single bulk synchronizer to all the threads created from a single bulk creation event. This bulk synchronizer acts as a semaphore. This modifies the synchronization on termination described in section 3.3.4: the event “request to synchronize” binds the bulk synchronizer, and not a single thread context, to a configurable register. Only when all associated threads terminate, does the bulk synchronizer cause the TMU to write a value to the target register, and let the waiting thread wake up. This allows a thread to wait on all threads in the group in a single register read. Beyond the reduction of thread management overheads for groups of related workloads, this facility also reduces fragmentation in the RF by grouping allocation and de-allocation requests in the RAU.

This particular feature which binds a set of threads by a common bulk creation event and a common bulk synchronization structure is another distinguishing trait of the proposed design. Groups of bulk created threads are referred to as “families” in previous publications; we discuss this term further in section 4.2.1.

3.4 Multi-core architecture

Placing multiple simple cores together is not sufficient to guarantee scalability of throughput with the number of cores. Namely, the interconnect must cater to scalable throughput (cf. section 2.2). Also, dedicated circuits must be present to provision low-latency distribution of data-parallel workloads to multiple cores, if these cores are to become an advantageous substitute to specialized SPMD/SIMD workloads (cf. section 2.4). We explain the corresponding characteristics of the proposed CMP design below.

3.4.1 Memory architecture

- The core design is optimized for tolerating latencies using asynchronous transactions with multiple in-flight operations, and is thus intended for use with memory systems that support either split-phase transactions or request pipelining. However, the concepts behind microthreading do not mandate a specific memory system: a designer may choose to integrate microthreaded cores with memory technology implemented independently. This was illustrated on a FPGA, where the UTLEON3 [DKK+12] microthreaded cores are connected to an industry-standard memory bus.

Regardless of which memory system is used, system-level design choices will impact the machine model significantly. First, an implementation can choose between a distributed memory architecture, where each cores sees a separate address space from all other cores. Or it might choose to use a shared address space. Then it can choose between different caching protocols. Depending on these choices, a memory model is shaped into existence.

The proposed architecture assumes a model based on a shared address space over multiple caches and backing stores. This in turn requires to consider a spectrum of design choices related to cache coherency. At one end of the spectrum an implementation may guarantee that all stores by all cores are visible to all other cores in the same order, for example by
Multiple cores can share a single L2 cache via a local snoopy bus. “DIR” is shorthand for “cache directory”; “MC” is shorthand for “memory controller.”

**Side note 3.4:** Using a distributed cache between microthreaded cores.

To demonstrate the usability of microthreading with larger core counts, a scalable memory system is needed. As a step in this direction, research efforts have been invested separately from our work in a dedicated synchronization-aware, distributed cache network. The main property of the proposed distributed cache is that cache lines are automatically migrated where they are used. Concurrent loads create multiple copies of a cache line, while concurrent stores may gather a single updated copy to the location of the last store. The proposed cache architecture connects multiple cores to a shared L2 cache, and organizes L2 caches in rings. For a small-scale system (e.g. less than 64 cores) a single ring is used; for larger configurations a hierarchy of rings is used (cf. fig. 3.8). An early version of the protocol has been published [ZJ07, DZJ08] and research is ongoing on this technology concurrently to our work. It assumes memory controller(s) to external Random-Access Memory (RAM) connected to the top-level ring, and pipelined RAM interfaces, such as JEDEC’s DDR\(^a\) and Rambus’ XDR\(^b\).

\(^b\)http://www.rambus.com/xdr

broadcasting a lock on the cache line before each store is effected and invalidating all copies. At the other end of the spectrum, no automatic consistency is implemented in hardware, so software must control cache coherency explicitly. Our reference implementation uses a memory network of distributed caches (side note 3.4) which lies somewhat in the middle: it proposes to effect memory stores in local copies of cache data, and only propagate and merge update copies prior to bulk creation of threads or bulk synchronization on termination. With this implementation, memory can be shared consistently across bulk events, but may not be consistent between sibling threads that are not related by bulk events.

Different implementation choices thus have a dramatic impact on the memory model. In chapter 7, we revisit this topic and further explore how to make these choices independent from concurrency management issues from the software perspective.
3.4. MULTI-CORE ARCHITECTURE

3.4.2 Concurrency management across cores

3.4.2.1 In other multi-processor systems

Work distribution across multiple performance-oriented general purpose processors has traditionally been coarse-grained and ill-suited for fine-grained, low-latency distribution of workloads. The conventional mechanism can be described in general terms: code and data are shipped to a memory close to the core at a fixed address; this address is configured in the interrupt vector of the target processor; then an Inter-Processor Interrupt (IPI) is sent to the remote processor to trigger execution. Synchronization on termination is expensive: the thread requesting to wait from core $A$ registers its identity in memory to a system-level scheduler in software; when a target thread on core $B$ terminates, its local scheduler looks up any waiting thread in memory, then sends an IPI from $B$ to $A$, which signals $A$’s software scheduler; $A$’s scheduler then looks up the origin of the IPI in memory and triggers resumption of the waiting thread.

This is the protocol used in most of the Top500\(^6\) and Green500\(^7\) supercomputers at the end of 2011, which are based on IBM’s POWER, Intel’s Xeon, AMD’s Opteron and Sun’s/Fujitsu’s SPARC6. It is used even between multiple cores on the same chip. It is also used with most processors in embedded SoCs, including all ARM and MIPS-based cores. This protocol is due to the processors’ ISA: in most ISAs only memory-related operations and interrupt instructions have the ability to communicate and synchronize with other components in the system.

This protocol is coarse-grained for three reasons. First, the overhead of a full point-to-point memory synchronization requires full cache lines to be exchanged between the initiator and the target, even if the target workload operates on independent streams of data. Then a scheduler in software on each processor must disambiguate each incoming IPI by looking up the initiator and parameters in shared structures in memory. Then the abstraction mismatch between the interrupt-based signalling mechanism and the thread-based programming model requires indirection through a software stack to effect the control events. This implies initialization and start-up overheads of hundreds if not thousands of pipeline cycles, and similar overheads again for synchronization on termination.

Incidentally, other hardware protocols for work distribution have existed and continue to be popular to this day. For example in IBM’s Cell Broadband Engine, the Memory Flow Controller (MFC) inside each Synergistic Processing Element (SPE) provides a hardware request queue where the Power Processor Element (PPE), a general-purpose PowerPC core, can write new thread requests remotely. The MFC then autonomously starts execution of the thread on the Synergistic Processing Unit (SPU) [GEMN07]. Another example is NVidia’s Tesla architecture. On this General-purpose GPUs (GPGPUs), the host-GPU interface streams units of work to the Texture/Processor Clusters (TPCs) via dedicated data paths in hardware. They are then decoded on the TPC and distributed to the Streaming Multiprocessors (SMs) via dedicated, autonomous SM controllers [LNOM08].

3.4.2.2 In the proposed architecture

Meanwhile, we have already explained in section 2.7 that further exploitation of multiple cores on chip will require lower concurrency management latencies. The proposed multi-core architecture design provides two features in this direction.

---

\(^{6}\)http://www.top500.org

\(^{7}\)http://www.green500.org
Side note 3.5: Sub-events for remote creations.

While delegation requests conceptually include all the parameters of a bulk creation, we will find useful to separate the phases in sub-events, namely requests for allocation of thread contexts and registers, remote register accesses to populate initial value in registers, bulk configuration requests to configure logical index ranges, and creation requests which trigger the actual activation of threads in the TCU.

The first feature is the ability of the TMU to generate active messages on the NoC, upon local reception of a delegation request control event from the pipeline. This complements the mechanism described in section 3.3.2 and fig. 3.7a: just as cores can receive remote thread creation events from the NoC, they can symmetrically generate those requests as well (cf. also side note 3.5). For synchronization on termination, the protocol from section 3.3.4 is also extended: a request for remote synchronization generated by the pipeline is forwarded by the TMU through the NCU and NoC, and is then received by the remote TMU which associates both the originating core address and register address to the target thread context. When the thread terminates, its local TMU sends a remote register write message through the NoC to the TMU of the processor hosting the waiting thread, which writes the value to the RF which in turn causes the thread to wake up. This feature enables fully general concurrent work creation and synchronization with an overhead of only a few cycles between adjacent cores on the NoC.

The other feature is an extension of bulk creation to support automatic distribution of threads across cores. In the TMU, an optional parameter can specify to restrict the number of logical thread indexes to create locally, and forward the remainder as a bulk creation request to another core. The total number of cores to use this way is also a parameter, or a feature of the NoC topology (which can be organized in static clusters). The thread creations are then effected locally by the TCUs in parallel. This way, a single bulk creation request sent to one core can trigger simultaneous thread creation on multiple cores. Similarly, a parameter can cause the TMU to duplicate automatically a register write request to all the cores participating in the bulk creation, implementing a broadcast operation. For synchronization on termination, the bulk synchronizers are chained from one core to another when the bulk creation request is forwarded, so that the bulk synchronization on the first core only completes when all “next” cores have synchronized locally. In short, this combination of features enables the automatic distribution of parallel workloads across cores using single requests to one core, including the parallelisation of thread creation itself. We describe cross-chip distribution further in chapter 11.

While setting up a coordinated group of threads across cores using this latter feature set may seem complicated, as we will see in chapter 4 this complexity is hidden behind simple abstractions.

3.4.3 Logical and physical networks

The previous sections have outlined three logical networks:

- the memory network, used for loads and stores issued by programs;
- the delegation network, used for program-generated remote concurrency management messages across arbitrary cores;
- the distribution network, used for TMU-controlled bulk distribution across neighboring cores.
### 3.4. MULTI-CORE ARCHITECTURE

**Figure 3.9: 32-core tile of microthreaded processors.**

White numbered tiles represent cores. “R” tiles represent routers for the narrow delegation mesh. The thick black link between cores represents the linear bulk distribution network.

<table>
<thead>
<tr>
<th>Event name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>thread creation</strong></td>
<td>PC, register window layout</td>
</tr>
<tr>
<td>(subsumed by bulk creation)</td>
<td></td>
</tr>
<tr>
<td><strong>request for synchronization</strong></td>
<td>target thread context, address of register to write to upon termination</td>
</tr>
<tr>
<td>(subsumed by bulk sync.)</td>
<td></td>
</tr>
<tr>
<td><strong>bulk creation</strong></td>
<td>common PC, common register window layout, overlap factor, logical thread index range</td>
</tr>
<tr>
<td><strong>request for bulk synchronization</strong></td>
<td>target bulk synchronizer, address of register to write to upon termination</td>
</tr>
<tr>
<td><strong>remote register reads &amp; writes</strong></td>
<td>target register address, value (for writes), address of register to send value to (for reads)</td>
</tr>
</tbody>
</table>

**Table 3.1: Control events to the TMU.**

These logical networks can be implemented either on top of a shared physical NoC, or using separate physical links for maximum throughput. For example, the reference configuration we use implements a dedicated distributed cache network for memory with cache-line-wide channels, a narrow mesh for delegations, and a linear word-wide array for bulk distribution, as illustrated in fig. 3.9. This specific network design uses a space filling curve to establish a multi-scale affinity between core addresses and caches, to preserve maximum data locality in bulk creations regardless of the number of cores involved.
### Table 3.2: Private state maintained by the TMU.

<table>
<thead>
<tr>
<th>State</th>
<th>Update events</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Program counters</em></td>
<td>Thread/bulk creation, branches</td>
</tr>
<tr>
<td><em>Mappings from logical register windows to</em></td>
<td>Thread/bulk creation</td>
</tr>
<tr>
<td><em>the register file</em></td>
<td></td>
</tr>
<tr>
<td><em>Logical index ranges</em></td>
<td>Bulk creation</td>
</tr>
<tr>
<td><em>Bulk synchronizers</em></td>
<td>Bulk creation, bulk synchronization</td>
</tr>
</tbody>
</table>

This state is maintained in dedicated hardware structures close to the TMU.

### Table 3.3: Logical sub-units in the TMU.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Thread Creation Unit (TCU)</em></td>
<td>Responsible for bulk thread creation and logical index distribution</td>
</tr>
<tr>
<td><em>Register Allocation Unit (RAU)</em></td>
<td>Responsible for dynamic allocation and de-allocation of register ranges in the RF</td>
</tr>
</tbody>
</table>

### Summary

The overall chip design proposed by the architects at the University of Amsterdam is a dataflow/von Neumann hybrid architecture with hardware multithreading, which uses a large register file as the synchronization name space for split-phase long-latency operations. The core architecture has been co-designed with a dedicated concurrency management protocol, implemented in hardware in a coordinating Thread Management Unit (TMU), for controlling hardware threads and composing cores into a many-core chip. We summarize its features in tables 3.1 to 3.3. It is programmed via input control signals, also named “concurrency management events.” These control signals are in turn generated by the pipeline in response to new machine instructions, or from a Network-on-Chip via active messages. We describe its programming interface further in chapter 4.

The architecture does not constrain the memory implementation but strongly suggests a scalable memory architecture with support for split-phase transactions and request pipelining. Research is ongoing to provide a distributed cache network to that effect. Remote concurrent work creation and synchronization is supported with hardware primitives for active messages and remote register accesses on the NoC. Bulk creation and synchronization is further optimized by providing low-overhead automatic distribution across multiple cores by the TMU.