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

Poss, R.C.

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 15

Conclusions and future work
—on the interplay between inner and outer questions.

It is pitch black. You are likely to be eaten by a grue.

Dave Lebling, via Zork

Contents

15.1 Design principles and opportunities ................................................. 236
15.2 Inner vs. outer questions: a retrospective ........................................... 237
15.3 Where to go from here? ................................................................. 240
15.1 Design principles and opportunities

When the design principles behind hardware microthreading were first proposed, the envisioned features with the largest impact on performance were the exploitation of dataflow synchronization over instruction operands to implement dynamic scheduling, the acceleration of thread creation and synchronization in hardware via elementary machine instructions, and the unification of single-core and multi-core thread management under a common protocol also supported in hardware [BJM96, BHJ06a]. The more concrete CMP architecture described in part I constitutes a subsequent effort to consolidate and integrate these features so as to enable their empirical evaluation.

As we discovered, the principles of hardware microthreading impact the hardware-software interface beyond the extension of registers with dataflow state bits and hardware support for thread creation and synchronization. In particular, the following extra features are visible from software:

- **configurable register windows**: the software can choose, upon the activation of a hardware thread, how many physical synchronizers (registers) are visible from the thread. This departs from previous ISAs where the number of registers is an architectural constant. The purpose of this change is to provide more efficient use of the register file by heterogeneous workloads.

- **active messages and asynchronous notifications via thread creation**: the reception of an external, asynchronous event by a processor core triggers the creation and activation of a new hardware thread, instead of the interruption of the control flow of an existing thread. This departs from previous ISAs where asynchronous events are delivered via interrupts. The purpose of this change is to simplify the individual core pipeline.

- **hardware synchronization services separate from main memory**: the extension of registers with dataflow state bits, together with the integration of a dedicated delegation/distribution NoC with the hardware scheduler on each core via the TMU, provides dedicated hardware support for I-variables, named critical sections, locking, barriers and semaphores. This departs from previous ISAs where synchronization semantics were provided as side-effects of ordering constraints on load and store operations to main memory. The purpose of this change is to provide lower synchronization overheads and a higher execution efficiency.

On their own, these extra features do not fundamentally interfere with existing ISA semantics. The system programmer can configure the register window of all threads to use a fixed number of registers, at the cost of a lower register file utilization, to obtain the fixed window semantics of previous ISAs. The designer of individual cores can integrate hardware microthreading in an existing RISC core without removing the interrupt delivery logic to preserve the preemption semantics of previous ISAs. The chip designer can integrate hardware microthreading in a CMP without removing support for consistency control in the caches and the memory network, to preserve the memory-based synchronization semantics found in existing CMPs. In theory, there is thus space in the design spectrum for a new CMP constructed from legacy RISC cores with hardware microthreading integrated as an extension. We depict this opportunity in fig. 15.1.

However, the researchers at the University of Amsterdam have made one further design step before investing in an implementation. They observed that asynchronous event delivery via thread creation is redundant with support for interrupts, and they observed that a dedicated synchronization protocol in hardware is redundant with memory support for
synchronization. Consequently, they decided to omit the corresponding support from their envisioned implementation of hardware microthreading, so as to simplify the hardware logic and thus increase execution efficiency further. The resulting CMP architecture, which we detailed in part I, is also depicted in fig. 15.1.

15.2 Inner vs. outer questions: a retrospective

The computer engineering steps to a product applicable to problems outside of computing science requires the participation of practitioners across multiple fields. In particular, any innovation at the level of individual components requires both to narrow down the substance of the innovation, that is what the new components look like, and to demonstrate its applicability in a larger system, that is how to integrate the new components. These are two phases of the engineering process, which we identify respectively in chapter 1 as the answers to the “inner” and “outer” question around innovation in computer architecture.

Previous work on hardware microthreading had focused mainly on answering the inner question. The publicly funded project from which this book originated, Apple-CORE, was an initiative to follow up with answers the outer question by multiple research organizations. As a member of this research community, we did not independently answer the inner nor the outer question around hardware microthreading; instead, we have:

1. recollected the collective work on the inner question (part I);
2. proposed enabling technology to help the collective answer to the outer question (chapters 6 and 8 to 11);
3. summarized the current collective work to the outer question (chapters 13 and 14).

The vantage point of the interface between platform provider and operating software providers provided two unique opportunities. First, we could identify several points where the answers to the outer question influence the answers to the inner question, i.e. where integration provides feedback on design. We recollect these in section 15.2.1. Then, by studying the consequences of removing features from the machine interface, we gained insights about what are the fundamental requirements of general-purpose computing and how they are affected by hardware microthreading. We comment on this in section 15.2.2.

15.2.1 Feedback on the inner question

By inviting operating software providers to interfere with the architecture design, we enabled a form of hardware/software co-design. For example, as we show in chapter 4, separate compilation in software engineering mandates a specific choice for the implementation of configurable register windows, which would be otherwise neutral from the architect’s perspective. More generally, as we argue in chapter 5, early feedback helps saving on development costs by avoiding design points which hinder further integration.

Beyond potential benefits in optimizing the overall engineering process, answering the outer question early also reveals additional requirements on the design of individual components. For example, as we show in chapter 9, the efficient provisioning of TLS, which is a logical feature of software, in turn requires ISA extensions and special support from caches. As we discuss in chapter 14, logical mutual exclusion considered as a software service requires to extend the proposed hardware synchronization protocol and may constrain the chip topology. As we argue in chapters 9 and 14, further support for process isolation and virtual memory addressing will require mechanisms for process identification in the hardware concurrency management protocol.

These observations confirm that combining integration with design is not merely a benefit to the platform provider; it is actually a necessity to avoid the HIMCYFIO pitfall we identified in chapter 1.

15.2.2 General-purpose computing and hardware microthreading

Beyond the productivity advantages of automated computing, the advent of general-purpose computers was a key step forward in the progress of mankind: it democratized the process of constructing solutions in software to both current and future problems at virtually no cost.

Meanwhile, serious upcoming technology challenges are the cause of a flurry of contemporary research activities around multi-core architectures and parallel programming. As we acknowledged in chapters 1 and 2, these challenges can be partially addressed by creating faster and more efficient chips tailored to specific applications. However we argue that any durable solution should strive to preserve generality in our computing artefacts. In the context of hardware microthreading, this issue is relevant because some features that make legacy processors fully general, namely the semantics of main memory and asynchronous event delivery, are altered in the proposed CMP design.

As a first step towards illustrating the generality of the design, we demonstrated that the resulting chip can run arbitrarily sequential computations described by a general-purpose
language, by constructing a C compiler and implementing minimal support for a C library (chapter 6). Towards asynchronous event delivery, we are able to use thread creation instead of control flow interrupts for external I/O (chapters 5 and 6). We discovered ground for further research towards supporting traps and exceptions (chapter 14), which we mention again below in section 15.3; however we believe that there are no further conceptual issues with claiming that spontaneous creation of logical threads constitutes a workable substitute to interrupt delivery.

In contrast, we discovered fundamental issues related to the interaction between cores and memory when using the proposed CMP for parallel computations. Beyond the requirement that multiple logical threads must each independently have access to TLS to fully simulate communicating processes where each process is Turing-complete, a topic which we explore in chapters 9 and 12, we discovered two major concerns.

Our first concern is that the generality of a parallel computing system, as opposed to a single-processor computer, is dependent on the ability to define arbitrary communication patterns between simultaneously running processing agents. By simplifying its individual cores and weakening its memory semantics, two choices motivated by potential reductions of hardware and energy costs, the designers of the proposed CMP have abandoned the memory-based mechanisms to define arbitrary parallel computations that had been devised in previous SMP architectures. To our knowledge, no prior work had acknowledged this differentiation as we do in chapter 7. To preserve generality while keeping the benefits of a simplified design, we propose in chapter 7 new semantics for programming languages to define arbitrary communication patterns over a weakly consistent memory and a dedicated synchronization protocol, such as found in the proposed CMP. Then, in chapter 12, we show how the proposed CMP could theoretically run programs assuming previous abstract concurrent execution models.

Our second concern is that software audiences rely on the ability to abstract hardware resources in programming languages, a requirement which mandates mechanisms to virtualize resources. As we argue in chapter 10, each resource type potentially needs different mechanisms. For storage and I/O, a well-know mechanism is the address space virtualization provided by address translation in hardware. This facility is largely preserved with hardware microthreading, although it receives a new interface and thus mandates some extra attention in operating software (chapters 5, 9 and 14). Meanwhile, the introduction of hardware microthreading implies the apparition of three new hardware resource types that now need to be virtualized: thread execution contexts, bulk synchronizers and dataflow synchronizers. For thread execution contexts, the machine interface already proposes thread virtualization by automatically multiplexing logical threads over hardware contexts (chapters 3 and 4). However, as we highlight in chapter 14, the coarser-grain activities spawned by multiple separate software applications sharing hardware resources may in turn mandate new mechanisms to virtualize entire multi-threaded processes (as opposed to single threads), which we outline at a high level in chapters 9 and 14. For bulk synchronizers, we argue in chapter 10 that the advent of declarative concurrency in programming languages obviates the need to virtualize individual bulk-created and bulk-synchronized “families” of logical threads. We have not yet found suitable mechanisms to virtualize dataflow synchronizers, and we suggest that a solution should be proposed in future work before hardware microthreading becomes fully accepted by software communities.
15.3 Where to go from here?

There are two perspectives to suggest future work.

- The first is the perspective of the innovator, or the research group(s) who will follow up on the inner question: research on the substance of hardware microthreading. By exposing the innovation to current software ecosystems, we revealed that the inner question has not yet been fully answered. Namely, key architectural features and research issues must be addressed before the design becomes fully suitable for use in applications:
  
  - the memory architecture must be fully defined, including its topology and address-to-bank mappings and virtual address translation mechanisms. As we discussed in chapters 7 and 9 and sections 13.7.1 and 14.3, the memory system’s characteristics will impact how run-time operating software will be constructed around the platform;
  
  - the architectural support for concurrency management must be complemented with support for asynchronous events and traps. Barring the reintroduction of control flow preemption, it is not yet clear how software can react to unforeseen circumstances in the run-time environment. As we highlighted in chapter 14, any programming language implementer for this platform will require a clear vision on this topic before it can be used to implement larger applications;
  
  - proper support for process boundaries must be introduced in the machine interface. We have specifically highlighted in chapters 6, 9 and 14 that resource reclamation (memory, cores, thread contexts) upon process termination requires the identification of resources that belong to different processes. Process boundaries are also needed to provide isolation and accountability, which we did not cover in our work but are expected from all software ecosystems. The current hardware interface does not yet address these aspects;
  
  - proper support for resource virtualization must be proposed and documented. While the machine interface proposes automatic virtualization of tasks as logical threads (chapters 3 and 4) and we could propose mechanisms to virtualize bulk synchronizers (chapter 10), no mechanism yet exists to fully virtualize the dataflow synchronizers and entire processes;
  
  - performance and resource models must be devised that establish a relationship between the architectural parameters and the behavior of programs. As we illustrated in chapter 13, naively implemented algorithms are able to obtain a comparatively higher throughput per unit of area than with legacy chips. However, performance scalability is actually constrained by a combination of parameters [GBK+09]: on-chip communication bandwidths, number of thread contexts per core, number of cores per L2 cache, etc. As we suggest in chapters 11 and 13, the implementation of space schedulers in operating software by third parties will only be possible once models can successfully predict the performance of program components running on different regions of the chip.

- The other perspective is that of the computing science community who will follow up on the outer question: developing the context where hardware microthreading can be applied. We foresee the following necessary steps:
  
  - while the research topics listed above are being studied, test and simulation platforms must be developed where the updated chip designs can be evaluated across a larger range of applications, including larger workloads. During our own work, we had access only to a low-level simulation of the multi-core design running at a few million
15.3. WHERE TO GO FROM HERE?

instructions per second and a single-core implementation on an FPGA. To our knowledge, this is a common limitation in most contemporary research projects in computer architecture. To support larger workloads during testing, including industry-standard benchmarks, faster simulators and emulators for many-core chips must be developed: either higher-level simulations in software running on distributed computational clusters, or by connecting multiple FPGA platforms via high-speed networks that mimic on-chip topologies;

- to support the diversity of existing software frameworks, further operating software must be developed to expose the generality we revealed in chapter 12. We could envision, for example, custom implementations of the POSIX threading library or ISO C11’s concurrency management primitives, an implementation of MPI, a back-end for an OpenCL compiler, or support in the run-time system of fashionable productivity languages like UPC or Chapel;

- an exploration of the architecture design space must take place to determine a balanced configuration of cache sizes, number of thread contexts per core, number of cores, number of synchronizers per core, and NoC characteristics for a given technology and area budget. This is necessary to eventually lay out the floorplan of a chip to estimate its power requirements prior to manufacturing.

We highlight that these steps are not specific to the design we presented in part I. They apply to any innovation in general-purpose microprocessor design that pushes for large-scale parallelism on chip.