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

Poss, R.C.

Citation for published version (APA):
Summary

Ever since the turn of the century, fundamental energy and scalability issues have precluded further performance improvements in general-purpose uniprocessor chips. To “cut the Gordian knot,” [RML+01] the industry has since shifted towards multiplying the number of processors on chip, creating increasing larger Chip Multi-Processors (CMPs) by processor counts, to take advantage of efficiency gains made possible by frequency scaling [RML+01, SA05]. Yet so far most general-purpose multi-core chips have been designed by grouping together processor cores that had been originally designed for single core, mostly single-threaded processor chips. After a decade of renewed interest in CMPs, the architecture community is barely coming to terms with the realization that traditional cores do not compose to create easily programmable general-purpose multi-core platforms.

Instead, the Computer Systems Architecture group at the University of Amsterdam proposes a general machine model and concurrency control protocol, relying on a novel individual core design with dedicated hardware support for concurrency management across multiple cores [PLY+12]. The key features of the design, described as “hardware microthreading,” are asynchrony, i.e. the ability to tolerate operations with irregular and long latencies, fine-grained hardware multithreading, a scale-invariant programming model that captures clusters on chip of arbitrary sizes as single programming resources, and the transparent performance scaling of a single binary code across multiple cluster sizes. Its machine interface does not only provide native support for dataflow synchronisation and imperative parallel programming; it also departs from the traditional RISC vision by allowing programs to configure the number of hardware registers available by thread, replacing interrupts by thread creation as a means to signal asynchronous events, relying on a single virtual address space, and discouraging the use of main memory as an all-purpose synchronization device, preferring instead a specialized inter-core synchronization protocol.

The adoption of a different machine interface comes at the cost of a challenge: most operating software in use today to drive general-purpose hardware, namely operating systems, programming language run-time systems and code generators in compilers have been developed with the assumption that the underlying platform can be modelled by traditional RISC cores with individual MMUs grouped around a shared memory that can be used for synchronization. A port of existing operating software towards the proposed architecture is therefore non-trivial, because the machine interface conceptually diverges from established standards. In this dissertation, we investigate the impact of these conceptual changes on operating software.
We propose namely answers to the following questions:

1. Is it possible to program a chip with the proposed machine interface using an already accepted general-purpose programming language such as C?
2. What are the abstract features of the proposed machine interface that make it qualitatively different from contemporary general-purpose processor chips from the perspective of operating software?

The first question is relevant because the availability of existing programming languages is a prerequisite for adoption of a new general-purpose architecture. Moreover, support for C must be available before most higher-level software environments can be reused. For this question our answer is generally positive. By constructing a C compiler and parts of the accompanying language library, we demonstrate that programs following the platform independence guidelines set forth by the designers of C can be reused successfully on multiple instances of the proposed architecture. We also demonstrate how to extend the C language with new primitives that can drive the proposed hardware-based concurrency management protocol. However, we acknowledge that most programs also use system services and make assumptions about the topology and components of the underlying platform. We discuss why some of these assumptions cannot yet be adapted fully transparently to the proposed architecture and suggest a strategy for future work to do so.

The second question is relevant because its answer defines how to advertise the platform to system programmers, who constitute the early technology adopters with the strongest influence. For this question our answer considers separately the various peculiarities of the machine interface.

The proposed ISA provides native hardware support for thread management and scheduling and thus seems to conflict with the traditional role of operating software. Yet as we argue this support does not change existing machine abstractions qualitatively, because concurrency management was already captured in operating software behind APIs with semantics similar to those of the proposed hardware protocol. The machine interface provides configurable numbers of registers per hardware thread, which is a feature yet unheard of in other general-purpose processors. Yet as we show this feature can be hidden completely behind a C code generator, and can thus become invisible to operating system code or higher-level programming languages. The proposed chip topology promotes a single address space shared between processes, relying on capabilities [CLFL94] instead of address space virtualization for isolation, which diverges from the process model of general-purpose operating systems commonly in use today. Yet as we suggest the technology necessary to manage a single virtual address space is already available and widely used (for shared libraries and application “plug-ins”) and this feature thus does not pose any new conceptual difficulty.

We found that the first conceptual innovation that warrants further theoretical investigation is the surrender of shared memory as the universal synchronization device for software. In traditional multi-core programming, implicit communication via coherent shared memory locations is routinely abused to provide locking, semaphores, barriers and all manners of time synchronization between concurrent activities. In the proposed architecture, such implicit communication is restricted and new basic programming constructs with fundamentally different semantics must be used instead. We formalize a subset of these semantics, then suggest how they can be used at a higher level in existing concurrent programming languages. To summarize, despite their strong conceptual divergence, we found that these
new forms of synchronization are fully general and can theoretically be integrated in existing software, although more work will be needed to actually realize this integration.

The second conceptual innovation we found is the finiteness of concurrency resources. In most implementations of multi-threading on general-purpose platforms, threads and synchronization devices (e.g. mutex locks) are logical concepts instantiated by software, with the assumption that they can be virtualized at will using main memory as a backing store. In the proposed architecture, threads and synchronization devices are finite resources which cause execution deadlock when programs attempt to create more of them than are available in hardware. We argue that thread virtualization is not necessary with the advent of declarative concurrency in programming languages. Unfortunately, we cannot determine whether finiteness of synchronization devices is a net benefit towards further adoption of the architectural concepts. To summarize, we found that this innovation will require further analysis and possibly further refinements of the proposed architecture.

Beyond the scientific contribution towards the two questions outlined above, this book also contains a narrative about the design and implementation of a new processor chip in the midst of contemporary technology challenges. In particular, we highlight that a common trend in processor architecture research is to “solve” conceptual issues by abandoning support for some programming styles and software abstractions. We condemn this trend as being detrimental to true general-purpose computing, and we discuss throughout our dissertation how the requirement to preserve generality in processors impacts development, from the architecture up to applications via operating software.