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

Citation for published version (APA):
Chapter 2

Trade-offs in microprocessor design

Abstract

Innovation by hardware architects takes place in a cultural context, the zeitgeist of the current technical age. To help audiences recognize the innovation as such, this background knowledge must be identified and communicated explicitly in a way that highlights current shortcomings. In this chapter, we propose such a perspective to motivate the innovation described in chapter 3 onward. By analyzing current trends and general trade-offs in CMP design, we identify the innovation space for microprocessors with more numerous, smaller general-purpose cores featuring HMT and hardware-supported concurrency management.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>34</td>
</tr>
<tr>
<td>2.2 Size matters</td>
<td>35</td>
</tr>
<tr>
<td>2.3 Sequential performance</td>
<td>36</td>
</tr>
<tr>
<td>2.4 Limits of specialization</td>
<td>37</td>
</tr>
<tr>
<td>2.5 Dynamic heterogeneity</td>
<td>38</td>
</tr>
<tr>
<td>2.6 Fine-grained multi-threading</td>
<td>39</td>
</tr>
<tr>
<td>2.7 Hardware concurrency management</td>
<td>40</td>
</tr>
<tr>
<td>Summary</td>
<td>41</td>
</tr>
</tbody>
</table>
CHAPTER 2. TRADE-OFFS IN MICROPROCESSOR DESIGN

2.1 Introduction

A condensed summary of the pre-2000 history of general-purpose microprocessor design is best quoted from [RML+01]:

In the past several decades, the world of computers and especially that of microprocessors has witnessed phenomenal advances. Computers have exhibited ever-increasing performance and decreasing costs, making them more affordable and, in turn, accelerating additional software and hardware development that fueled this process even more. The technology that enabled this exponential growth is a combination of advancements in process technology, micro-architecture, architecture, and design and development tools. While the pace of this progress has been quite impressive over the last two decades, it has become harder and harder to keep up this pace. New process technology requires more expensive megafabs and new performance levels require larger die, higher power consumption, and enormous design and validation effort. Furthermore, as CMOS technology continues to advance, microprocessor design is exposed to a new set of challenges. In the near future, micro-architecture has to consider and explicitly manage the limits of semiconductor technology, such as wire delays, power dissipation, and soft errors.

The authors of this paper detail the obstacles faced by architects on the way to faster and more efficient processors. A summary can be found in [BHJ06a]: fundamental energy and logic costs hinder further performance improvements for single instruction streams. To “cut the Gordian knot,” in the words of [RML+01], the industry has since (post-2000) shifted towards multiplying the number of processors on chip, creating increasingly larger CMPs by processor counts, now called cores. The underlying motivation is to exploit higher-level parallelism in applications and distribute workloads across multiple processors to increase the overall throughput of computations\(^1\).

This shift to multi-core chips has caused a commotion in those software communities that had gotten used to transparent frequency increases and implicit Instruction-Level Parallelism (ILP) without ever questioning the basic machine model targeted by programming languages and complexity theory. “The free lunch is over” [Sut05], and software ecosystems now have to acknowledge and understand explicit on-chip parallelism and energy constraints to fully utilize current and future hardware.

This may seem disruptive when most textbooks still describe computers as a machine where the processor fetches instructions one after another following the control flow of one program. Yet this commotion is essentially specific to those traditional audiences of general-purpose processors. In most application niches, application-specific knowledge about available parallelism has long mandated dedicated support from the hardware and software towards increased performance: scientific and high-performance computing have long exploited dedicated Single Instruction, Multiple Data (SIMD), Multiple Instruction, Multiple Data (MIMD) and Single Program, Multiple Data (SPMD) units, embedded applications routinely specialize components to program features to reduce logic feature size and power requirements, and server applications in datacenters have been optimized towards servicing

\(^1\)While parallelism can also be used to reduce the latency of individual programs, Amdahl’s law gets in the way and Gustafson’s law [Gus88] suggests that the problem sizes of the parallel sections instead expand to the parallelism available at constant latency.
2.2 Size matters

Once the architect considers a CMP design, an opportunity exists to choose how much logic to invest per core vs. how much logic to invest towards a larger number of cores.

Once concurrency is available in software, it becomes advantageous to scale back the number of transistors per core and increase the number of cores. There are two reasons for this. The first reason is that the logic and energy costs invested in individual cores towards improving sequential performance, namely larger branch predictors, larger issue

\[2\text{Amdahl explained in [Amd67] that the performance of one program will stay fundamentally limited by its longest chain of dependent computations, i.e. its critical path, regardless of how much platform parallelism is available.}\]
width, duplicated functional units, deeper pipelines, larger reorder logic, etc., scale disfavorably with the Instructions Per Cycle (IPC) gains (e.g. two times more transistors does not increase IPC by two). Pollack summarized this situation in [Pol99] by stating that the performance of a sequential processor improves with the square root of the number of transistors. The other reason is that power consumption grows as a super-quadratic function of frequency [RML+01, SA05]: if a design enables throughput scalable with the number of cores, energy efficiency gains can be expected by running more cores at a reduced frequency (e.g. 10 cores at 200MHz each instead of 1 core running at 2GHz will consume less power, for the same maximum IPC). We illustrate this design spectrum in fig. 2.1.

2.3 Sequential performance and the cost of asymmetry

We suggest above that “smaller is better” and that many small cores will fare better overall than few larger cores in our emerging era of ubiquitous concurrency in software. Yet we should acknowledge that some inherently sequential workloads will still matter in the foreseeable future, both from legacy software and those applications where no parallel or distributed algorithms are yet known. To support these while still taking advantage of the available software concurrency, the processor architect has two recourses. The conservative approach is to favor homogeneity and slide the design cursor more to the left in fig. 2.1. This is the approach taken e.g. with the Niagara T4 [SGJ+12]. This simplifies the machine model exposed to programmers, but comes at the cost of less efficiency for more concurrent workloads.

The other approach is to introduce static heterogeneity and allocate some areas of the chip towards throughput and others towards sequential IPC. This is the approach taken e.g. with the AMD Fusion architecture [Adv, where “accelerator” cores are placed next to general-purpose cores on the same die. Our illustration of the corresponding spectrum of possible heterogeneous designs in fig. 2.2 reveals a possible pitfall: the appearance of model asymmetry as an historical artifact.

Indeed, the shift towards more on-chip parallelism has emerged from a background culture where a chip was a single processor. The availability of on-chip parallelism may thus appear as an extension of the well-known single processor. However, if a CMP is considered as a mostly-sequential processor with optional “parallel accelerators,” this will encourage
2.4 Specialization is an orthogonal, limited process

Another form of static heterogeneity already in use is one of function: when specific computations are provided by dedicated hardware logic. For example, a hashing unit to support cryptography and a Floating-Point Unit (FPU) for arithmetic are fixed-function components. The motivation for such specialization is the increased throughput per unit of area and per watt for the functions specialized and the algorithms that use them.

This opens a spectrum of designs which we illustrate in fig. 2.3. When a chip is dedicated to a given application scenario, and the scenario has been profiled, it becomes meaningful to provision the main computational components of the scenario using specialized hardware. Conversely, a given selection of specialized features is specific to the scenario considered, and may not match another scenario which uses different computational components.

In general, feature specialization in a hardware design is an orthogonal way to increase efficiency, i.e. increase overall application throughput at constant cost or reduce cost at constant throughput, for given application scenarios.

In contrast, a general-purpose chip is by definition not specialized towards specific scenarios. While one may be tempted to suggest mandatory specialized circuits for those few features that have become prevalent across most fields of computing, for example floating-point arithmetic, cryptography kernels and signal processing SIMD units for audio/video stream conversion, the question would still exist of how much logic to invest into these specialized units as opposed to e.g. more general-purpose cores, larger on-chip memories or a faster interconnect. Again, the proper methodology to strike the right balance is to consider the profile of contemporary applications at the time the overall chip design is decided and specialize for that.

Here we are able to recognize a limit on specialization.

---

Figure 2.3: Choice between specialized functions or general-purpose cores at equal logic and energy budget.
In general-purpose applications, workloads enter and leave the system at unpredictable times. The sharing of hardware components between workloads thus requires on-line, dynamic chip resource management. To satisfy the need for allocation times within the scale of operation latencies, resource management must be supported in hardware [KJS+02]. Since the amount of state that can be maintained locally on chip is limited, the component models used by on-line resource managers must be kept simple, which in turn implies that the diversity of component properties is kept low. An example of this can be found in the replacement of multiple bus hierarchies in larger CMPs and Systems-on-Chip (SoCs) by standardized low-latency protocols on a common packet-switched NoC [HWC04]. Moreover, once application requirements increase in complexity, for example when starting to account for time/energy budget allowances [MMB07], the pressure to reduce component diversity to keep on-line resource managers fast increases further.

These observations push chip designers in two directions. One calls for the full integration of re-configurable logic, e.g. FPGAs, in general-purpose chips, so that functions can be specialized on demand. Unfortunately, practitioners have not yet come up with update protocols that can perform reconfiguration at a fast rate. The other direction pushes for adaptive general-purpose cores connected by a general-purpose NoC, which simplifies on-line resource management by making pools of resources fungible. For example, a group of many in-order RISC cores on a mesh interconnect with configurable frequency and voltage may be an advantageous replacement for a fixed-frequency specialized SIMD unit with dedicated data paths, because it is reusable for other purposes without the overhead of hardware reconfiguration.

In short, while the throughput/cost ratio may be higher for isolated workloads running on specialized, monolithic units, fungibility of function is more desirable for dynamic, distributed and unpredictable workloads: it increases predictability of cost and responsiveness by reducing contention, either on the specialized components themselves or on resource managers. A recent acknowledgement of this view can be found in [SCS+08].

Note that favoring general-purpose units does not imply mandating homogeneity in chip designs: each unit can be configurable dynamically based on load and application requirements, at a fine-grain, with different throughput and cost parameters; frequency and voltage are examples. Besides constraints on on-chip resource management, outlined above, there are other simpler reasons to invest logic towards larger numbers of general-purpose cores instead of specialized functions. For one, it simplifies code generators by reducing the diversity of operation interfaces in the Instruction Set Architectures (ISAs). It also increases opportunities for fault tolerance by allowing more replacement candidates for faulty units. Finally, it reduces hardware design costs by allowing tiling.

2.5 Dynamic heterogeneity and unpredictable latencies

Another trend that supports re-configurable or fungible computing resources on chip is the increasing number of faults as the density of circuits and the number of transistors increases.

Both transient and permanent faults can be considered. Transient faults are caused mostly by unexpected charged particles traversing the silicon fabric, either emitted by atomic decay in the fabric itself or its surrounding packages, or by cosmic rays, or by impact from atmospheric neutrons; as the density of circuits increases, a single charged particle will impact

Fungibility is the property of a good or a commodity whose individual units are capable of mutual substitution. Examples of highly fungible commodities are crude oil, wheat, precious metals, and currencies.
more circuits. Permanent faults are caused by physical damage to the fabric, for example via heat-induced stress on the metal interconnect or atomic migration. While further research on energy efficiency will limit heat-induced stress, atomic migration unavoidably entails loss of function of some components over time. This effect increases as the technology density increases because the feature size, i.e. the number of atoms per transistor/gate, decreases.

To mitigate the impact of faults, various mechanisms have been used to hide faults from software: redundancy, error correction, etc. However, a fundamental consequence of faults remains: as fault tolerance kicks in, either the latency changes (e.g. longer path through the duplicated circuit or error correction logic) or the throughput changes (e.g. one circuit used instead of two).

To summarize, the increasing number of faults is a source of unavoidable dynamic heterogeneity in larger chips. Either components will appear to software to enter or leave the system dynamically, for example when a core must stop due to temporary heat excess, or their characteristics will appear to evolve over time beyond the control of applications.

This in turn suggests a distributed model of the chip (independently of the suggestion from section 2.2) which can account for the transient unavailability or dynamic evolution of parts of the chip’s structure.

### 2.6 Fine-grained multi-threading to tolerate on-chip latencies

The increasing disparity between the chip size and the gate size causes the latency between on-chip components (cores, caches and scratchpads) to increase relative to the pipeline cycle time\(^5\). This divergence is the on-chip equivalent of the “memory wall” [WM95]. This causes increasing mandatory waiting times in individual threads. Moreover, these latencies will be unpredictable, due to overall usage unpredictability in general-purpose workloads, due to faults as explained above in section 2.5, and due to feature size variability as the gate density increases [BFG*06]. These latencies must be tolerated by overlapping computations and communications within cores before the expected throughput scalability of multiple cores can be achieved.

These latencies cannot be easily tolerated using superscalar issue or VLIW, for the reasons outlined in section 2.2 and [BHJ06a]. To summarize, increasing the amount of concurrency in single-threaded cores via superscalar execution mandates non-scalable complexity in coordination structures like register files. In turn, VLIW is sensitive to mispredicted branches and variable memory latencies, and thus requires energy-inefficient speculation to maximize throughput.

An alternative to exploit mandatory waiting times of individual threads is Hardware Multi-Threading (HMT), i.e. the interleaving of fine-grained threads in the cores’ pipelines via a hardware scheduler. Note that hardware multithreading does not increase the sequential performance of individual threads; instead, it makes a system generally more efficient (less wasted time/energy), and increases overall throughputs when algorithms can successfully fill waiting times of some threads by useful work from other threads without increasing the length of the overall critical path.

The general principle of interleaving multiple threads through a single processor with smaller time slices than the communication latency they should tolerate is not new [Bem57,}

---

\(^5\) At constant wire length, the sections above suggest more smaller cores at lower frequency, which would imply less communication delay relative to the cycle time. However, the wire delay increases relative to the transit time across a gate; the latter in turn constrains frequency and puts a lower bound on pipeline cycle time.
However, as the ratio between average on-chip latencies between components and pipeline cycle time grows, all inter-component events become candidate for latency tolerance, including memory operations, floating-point operations and inter-thread synchronization events. In this setting, the latency overhead of software-directed preemptive interleaving of threads over a single physical execution context, i.e. the cost of saving and restoring PC and per-thread registers, could be too large compared to the latencies to be tolerated; therefore, hardware-directed scheduling over multiple physical thread contexts must be used if all latencies must be tolerated.

There have been previous successful approaches to implement HMT in general-purpose designs. On barrel processors (e.g. on the CDC 6600 [Tho65] where it was first implemented, and later with Denelcor’s HEP [Smi81]), the hardware scheduler assigns equal time slots to all threads. The MTA [BCS+98, SCB+98] follows up on the barrel processor concept and makes scheduling dynamic: only instructions from threads ready to execute enter the pipeline. On Simultaneous Multi-Threading (SMT) superscalar processors [TEL95, MBH+02], unused issue slots are filled by instructions from a secondary Program Counter (PC).

There are two issues with previous approaches to HMT however. One is throughput flexibility: the overall throughput should be divided more or less equally between active threads. This is an issue with pure barrel processors: with \( N \) contexts, the performance of each threads is \( \frac{1}{N} \) even if there are less than \( N \) threads active. This has been addressed in the MTA’s dynamic schedule queue, and is naturally not an issue in processors primarily designed for sequential performance. The other is robustness to branches: with long pipelines, the cost of mispredictions (or branches taken if there is no prediction) is high, as the pipeline must be flushed. This is an issue with the 21-stage MTA pipeline and most contemporary SMT superscalar designs.

These observations suggest that the benefits of HMT are anticipated to be apparent with shorter pipelines and dynamic schedule queues.

2.7 Pressure for hardware-assisted concurrency management

Assuming CMPs with an increasing number of cores and per-core HMT, space scheduling must be implemented to spread concurrent software workloads to the chip’s parallel execution resources. Space scheduling can be done either in software or in hardware.

With a software scheduler, each hardware thread is controlled by a program that assigns tasks using state taken from main memory. Specifically, each hardware thread is controlled by an instance of a software scheduler in an operating system, which is notified, via interrupt signalling and memory-based communication, upon task creation and begins execution until a termination or suspension event. This can occur even when thread interleaving is performed at a fine grain in hardware. It is also relevant even when there is no need for time sharing of multiple tasks onto a single hardware thread, for example when the number of tasks is smaller or equal to the number of hardware threads, or when cooperative scheduling is sufficient. However, the choice of a software scheduler assumes that the workload per task is always sufficient to compensate the non-local latencies incurred by memory accesses to task state during schedule decision making and task assignment.

This assumption traditionally holds for coarse-grained concurrency, for example external I/O. It can also hold for regular, wide-breadth concurrency patterns extracted from sequential tight loops, via blocking aggregation (e.g. OpenMP). However the situation is
not so clear with fine-grain heterogeneous task concurrency. For example, graph transformation and dataflow algorithms typically expose a large amount of irregularly structured, fine-grained concurrency. In these cases, a strain is put on compilers and run-time systems: they must determine the suitable aggregate units of concurrency from programs that both optimize load balancing and compensate concurrency management costs.

This motivates the acceleration of space scheduling, considered as a system function, using dedicated hardware logic. This idea to introduce hardware support for concurrency management is not new; it was pushed by researchers until twenty years ago [Smi81, HF88, NA89, MS90, CSS+91]. Back then, it met with resistance against the introduction of explicit concurrency in applications. Now that on-chip software concurrency is the norm, hardware support deserves renewed attention for two reason.

One is the potential gain in resource fungibility obtained by the replacement of specialized SPMD/SIMD units by multiple general-purpose cores (cf. section 2.4). For this to be tractable, the overhead to dispatch an SPMD task over all participating pipelines must be comparable to or smaller than the latency of the operation, e.g. a couple dozen pipeline cycles for most SPMD workloads. To make general-purpose cores an attractive substitute to specialized SPMD/SIMD units, extra hardware support must exist with low-latency bulk work distribution and synchronization.

The second argument is cost predictability: when a software scheduler is involved, it competes with algorithm code for access to the memory components. The overhead of communication and synchronization between software schedulers for task management increases with the number of hardware threads and interferes with communication for computations, introducing jitter [ALL89]. This can be avoided by a dedicated task control network separate from the memory network.

We should acknowledge here that hardware task and thread management somewhat reduces flexibility in the definition of a task from the perspective of software. As noted in [LH94], “thread definitions vary according to language characteristics and context-switching criteria.” We should not consider this argument to be a ban on hardware support for concurrency management, however. In fact, it seems to us desirable to provide some form of general hardware/software interface for the definition of concurrent workloads, so that hardware components with different designs can implement them in a heterogeneous CMP without creating asymmetry in the machine model exposed to software.

**Summary**

In this chapter, we have identified the design trade-offs available to a CMP designer in the age of ubiquitous software concurrency. Our analysis suggests chip designs towards smaller, simpler general-purpose cores where larger cores optimized towards sequential performance serve as a service to the rest of the system. We have also identified the applicability and limitations of hardware specialization of specific application features. We recognized Hardware Multi-Threading (HMT) as a means to tolerate unpredictable on-chip latencies and identified the renewed relevance of hardware-supported concurrency management for on-chip parallelism. This argument updates and extends previously published justifications [BJM96, BHJ06a] for architecture research towards hardware microthreading.