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

Poss, R.C.

Citation for published version (APA):
Chapter 5

System perspective
—Identifying audiences, their expectations and opportunities

Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime. Unless he doesn’t like sushi—then you also have to teach him to cook.

Auren Hoffman, cited in [Dac06]

Abstract

The responsibility of the hardware architect extends beyond the design of hardware components on a processor chip, to include meeting the platform expectations of software ecosystems. The reward for entering a dialogue with the audience is the opportunity to gain early feedback upon the innovation. In this chapter, we identify candidate ecosystems for the proposed design from part I, the parties involved in the dialogue and their expectations. This identification allows us to sketch a technical path, which we will then follow throughout part II.

Contents

5.1 Introduction .......................................................... 84
5.2 Computing ecosystems ............................................. 85
5.3 Proposed dialogue scenario and opportunities ............... 87
5.4 Case study: hardware microthreading .......................... 88
5.5 Platform definition ................................................... 92
Summary .................................................................. 95
5.1 Introduction

Ever since the advent of hardware specification languages like Verilog or VHDL, innovators in hardware architectures have been able to separate the design of new parameterized components from the selection of actual parameter values.

This raises the question of how to evaluate a new design. In principle, it is possible to look at a parameterized specification in isolation and analyse the impact of design choices locally. For example, the authors of [BJM96] study analytically the optimal number of threads that must be active simultaneously in a microthreaded core to tolerate memory latencies; this number of threads is then itself parameterized by the actual memory latencies in a concrete system. However, local study of a new component is only possible by making assumptions about the context where the component will be used. For example, simulating a new branch predictor using precomputed execution traces assumes that the sequence of instructions on a processor without the branch predictor is the same as when the branch predictor is available. In general, simulations of a new component in isolation are only valid if the models of the component’s environment, which serve as input to the simulation, take into account the impact of the innovation on the entire system.

The question of evaluation thus becomes non-trivial when a new component functionally impacts multiple levels of a computing system. Impact on function implies that the structure of software may become different, and therefore that models of system behavior constructed without the component become inaccurate as input to predict the behavior of a system equipped with the new component. In this case, the construction of a system, or at least a full system simulation, becomes unavoidable to evaluate the innovation.

This in turn raises the question of how much of a system one should construct to expose and evaluate the innovation. Processor chip designers typically consider that innovation in processor components is fully realized once the processor chip is fully configured, either on silicon or in simulations. Yet the impact of an innovation at the level of processors can only be studied in the context of software workloads. These workloads in turn are highly dependent on the environment of the processor, including memory, of course, but also:

- the logic that define I/O interfaces, e.g. an Ethernet adapter, which defines the protocol used by software to react to external asynchronous events;
- the first level code generation tools that provide a programming environment sufficient to implement software operating systems, since the operational semantics of the first level programming languages will constrain those of any higher-level language implemented on top of them.

There are thus two ways to expose a new component in processors, depending on the impact of the innovation on the software ecosystem. If existing I/O infrastructure and abstractions, as well as first level programming interfaces, can be reused without significant changes, then it becomes possible to argue that the new design can be used as a “drop-in replacement” in existing systems, i.e. that it “preserves backward compatibility.” It then becomes possible to use existing software and evaluation methods soundly, at a low cost. In contrast, if the new design requires changes to any of these aspects, then the party in charge of the innovation must take the responsibility to either implement the corresponding new I/O infrastructure or code generation technology, or partner with third-parties to provide them, before describing how software workloads must be adapted to the new proposed platform.

This is the topic introduced in this chapter, and covered in the rest of part II: we propose a methodology to support the exposition and realization of an innovation in processor chips, when the innovation has a potential impact on the system interface to software.
In section 5.2, we expose the concept of computing ecosystem and we highlight the interactions between ecosystem actors in the design of computing platforms. Then in section 5.3, we outline our methodology where the expectations and requirements of existing computing ecosystems are used as a starting point for integration. Our key idea is conceptual backward compatibility, i.e. the match of existing expectations in new integrations. This idea stems from the observation that hardware and software components are nowadays modular, and that a new platform provider can co-redesign part of the hardware platform and the software platform to tune it to a new chip design while reusing most components of existing systems. The benefit of this approach is that any form of partial technology reuse can provide early feedback on the substance of the innovation. Our methodology involves first selecting a computing ecosystem, then understanding their socio-technical dynamics, and in particular identifying the subset of their technology that interfaces software with hardware. Then we propose to use this understanding to derive integration steps where only this technology subset is replaced by functionally equivalent hardware and software components. We then apply this methodology in section 5.4 to the innovation from part I and derive possible integration steps. We outline in section 5.5 the resulting system platform, which was used subsequently for software developments.

5.2 Computing ecosystems

In any programming environment, application developers use external services in addition to language intrinsics. These give access to features outside of the core language semantics.

From the language implementer’s perspective, there is an additional qualitative distinction between autonomous library services which capture common algorithms or program patterns as sub-programs or functions expressed only using the language and other library functions; and system services which requires specific knowledge about the underlying computing system. For example in C the string comparison function `strcmp` is autonomous, whereas the `malloc` heap allocation function is a system service because it either requires to know about the address space layout of the actual hardware, or requires support from a virtual memory manager like POSIX’s `sbrk` or `mmap`. This distinction is useful because the definition of autonomous services requires only a dialogue between the application developer and the language implementer, whereas the definition of system services needs a dialogue between both these parties and the underlying system’s provider.

Which parties are involved, and how many, between the language implementers and the hardware provider constitutes a distinguishing feature of a computing ecosystem. For example, with the first High Performance Computing (HPC) ecosystems and the early personal micro-computer markets, there was no intermediate party between the language implementer and the hardware provider. The latter provided detailed knowledge about the hardware interface to both language implementers and application developers. This situation still exists with small embedded systems, especially those based on microcontrollers. In contrast, the IT service provider ecosystems of the early 21st century feature tens of parties involved in defining abstractions between the hardware platforms and the actual applications visible to customers.

5.2.1 Dialogue within ecosystems

The main dialogue between an ecosystem’s actors considered here is the user-provider relationship, illustrated in fig. 5.1.
CHAPTER 5. SYSTEM PERSPECTIVE

Figure 5.1: User-provider relationship in a computing ecosystem.

Figure 5.2: Idealized, non-realistic vision of the “abstraction stack.”

Figure 5.3: Subset of actual ecosystem interactions in the IT service industry.

(a) User-provider dialogues. The strength of the arrows represents the amount of knowledge provided.

(b) Run-time interactions between technologies.
In this dialogue, a user party expresses wishes, or expectations, which are answered by a provider party in the design and implementation of their technology. The provider’s design is then described to the user, who can subsequently use this knowledge to implement more technology. A computing system results from an actual composition of the technologies. Run-time interactions between components are only eventually possible if the dialogue between the parties was successful.

It may be tempting to describe a computing system as a linear, stacked relationship between components of increasing levels of abstractions (fig. 5.2). In this idealized vision, the actors in the ecosystem form a chain where the actor at one level of abstraction plays the role of user for the actor at the level immediately lower, and provider for the actor at the level immediately higher.

In reality however, ecosystems are more complex and a strict abstraction stack cannot be clearly defined. The example in fig. 5.3 demonstrates that some actor pairs are both user and provider to each other. For example, C compiler and library implementers are thus typically co-dependents. Moreover, some actors will place expectations and require knowledge from multiple providers. For example application programmers may place expectations, and require knowledge from, both the compiler and library implementers.

Despite this diversity, we can recognize in most ecosystems:

- the platform providers who have a privileged role as universal providers, as they do not require knowledge about the technology of other actors to provide their own technology (although they may advantageously exploit such knowledge when it is available);
- various operating software providers who do require knowledge about the platform and whose technology is not directly visible by the eventual external users of the running computing systems.

5.2.2 The need for a target audience

When pushing innovations from the hardware designer’s perspective, which really is a part of the platform provider’s perspective, it is thus essential to determine or choose a target ecosystem, so as to identify which parties are responsible for providing the first-level interface to the platform. Depending on the ecosystem, these first level parties will be either application-level programming language implementers, operating system providers, platform virtualization providers, or possibly others, all of whose will have different expectations and customs that must be acknowledged in the dialogue.

This is the obstacle that creates the HIMCYFIO pitfall introduced in chapter 1: often innovators resist choosing a target ecosystem, and thus fail to tailor their dialogue to the specific culture of their first level parties who have direct access to the innovation in the abstraction stack.

In this chapter, we propose a dialogue scenario to advertise architectural innovations, and we illustrate this scenario with the case of hardware microthreading introduced in part I.

5.3 Proposed dialogue scenario and opportunities

The integration of architectural innovations involves both gaining the interest of the various stakeholders and sketching an integration path for them. This must outline which concrete actions are required to perform the integration and fit the innovation into the ecosystem.
For this, two different perspectives must be adopted simultaneously. The first must look backwards in time and provide “adequate” support for legacy applications. This must acknowledge the perception of existing user communities, their assumptions and their culture. What constitutes “adequate” backward compatibility can be negotiated based on contemporary expectations: for example, the simple preservation of standard system Application Programming Interfaces (APIs) instead of full binary compatibility has become acceptable for commodity computing in the first decade of this century, whereas it was frowned upon ten years earlier. The second perspective must look forward and define an attractive toolbox for new applications. This must highlight issues with past customs of the ecosystem and illustrate new opportunities to gain traction. This is where a new feature can be advertised, for example hardware support for concurrency management.

This latter perspective is well-understood and usually spontaneously adopted by innovators. However, the first perspective should not be adopted as an afterthought: innovators who may resist acknowledging the full scope of their audience in the ecosystem may not recognize that it is their responsibility to sketch the integration path as an extension of current views.

To sketch an initial integration path, we suggest in a first iteration to mingle with the community and imitate previous integration scenarios from other successful innovators. We have three motivations for this.

One is that using a known scenario makes it easier for parties to recognize and estimate the work required to adopt an innovation. For example, porting an existing operating system to a new hardware platform is a well understood undertaking, whereas designing an entirely new operating system and then its backward compatibility layer involves high risks in management and planning.

Another is that reusing an existing scenario allows audiences to easily differentiate the innovation from its extra integration steps. For example, while porting an operating system to a new hardware platform, the new hardware features will be readily recognized by comparison with the other platforms where the operating system was running previously; whereas if, to expose a new hardware platform, a new operating system is developed, the benefits of the former may be occluded by design or engineering defects in the latter.

Our third motivation, perhaps the strongest, is that reusing a known scenario will minimize the effort to reach the first interested parties and encourage them to provide early feedback. This is essential as early feedback may highlight key issues with the innovation that must be addressed before effort is invested into a larger, long-term integration plan.

5.4 Case study: hardware microthreading

5.4.1 Target ecosystems

We have examined possible ecosystems for the innovation from part I, summarized in table 5.1. This table identifies for each ecosystem the parties interacting directly with the platform provider and evaluates the estimated acceptability of the proposed hardware features.

From this analysis we can recognize an interesting target: the community of IT service providers, more specifically type II and III service providers who build upon Free and Open Source Software (F/OSS).

---

1As per [RH11]: type I service service providers are those providing IT services to only one client,
5.4. CASE STUDY: HARDWARE MICROTHREADING

Ecosystem ↓
First level parties in direct dialogue with the hardware provider
Preference for homogeneous, fungible chip structure rather than heterogeneous, specialized components
Tolerance of changes to the machine interface, forcing recompilation of programs

<table>
<thead>
<tr>
<th>Ecosystem</th>
<th>First level parties in direct dialogue with the hardware provider</th>
<th>Preference for homogeneous, fungible chip structure rather than heterogeneous, specialized components</th>
<th>Tolerance of changes to the machine interface, forcing recompilation of programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC</td>
<td>Both performance language implementers and application providers</td>
<td>usually increasing</td>
<td>increasing</td>
</tr>
<tr>
<td>Desktop computing</td>
<td>Both application providers and operating software providers</td>
<td>yes</td>
<td>unusually, but increasing</td>
</tr>
<tr>
<td>Mobile computing</td>
<td>System integrators</td>
<td>unusually</td>
<td>yes</td>
</tr>
<tr>
<td>Video game consoles</td>
<td>Game designers</td>
<td>unusually</td>
<td>unusually</td>
</tr>
<tr>
<td>Embedded systems</td>
<td>Application providers</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Type II/III IT service providers</td>
<td>Operating software providers</td>
<td>yes</td>
<td>unusually</td>
</tr>
<tr>
<td>Type II/III IT service providers</td>
<td>Operating software providers</td>
<td>yes</td>
<td>usually</td>
</tr>
</tbody>
</table>

Table 5.1: Possible target ecosystems for the proposed architecture.

The candidate ecosystems are taken from those most likely to benefit from the availability of large amounts of parallelism on chip. This excludes e.g. domestic appliances.

Type II/III service providers are particularly interesting for two reasons. First, this community does not typically expect specialized computing platforms, i.e. its members expect balanced general-purpose platforms that they can specialize differently in software for different customers. This contrasts with type I providers who may consider specializing to the needs of their unique client. Type II/III providers are thus direct consumers of the “stem cells” of computing that we recognized in section 1.3; this ecosystem is therefore likely to be accepting of, or interested in, hardware designs that are not tailored to a specific application.

The second reason is that this ecosystem’s market dynamics do not allow products to crystallize into aging legacy that must be preserved and maintained over time. This is because the lifetime of the associated business contracts does not typically exceed the lifetime of the platforms used. To the contrary, parties in this ecosystem welcome most technology developments as a positive factor for business differentiation. They are thus accepting of significant platform developments, e.g. new ISA features that require recompilation of program code, new filesystem technology that requires data migration, etc. This is useful since the proposed architecture does not propose to maintain backward binary compatibility of application code with previous platforms.

The subset of F/OSS-oriented service providers [GA04] is also particularly interesting for two reasons. The first is that the F/OSS movement is now widely acknowledged as a driving force for innovation in software ecosystems [Lev84, Web00, LT02, vHvK03, LW03, LT04, BS06, CHA06, Rie07]. Moreover, in the first decade of this century, F/OSS has also become the primary source of first level infrastructure for the online IT service industry. It must thus be acknowledged by hardware architects who want to partake in horizontal innovation networks [vH07]. The second reason is that F/OSS, by definition, promotes openness and transparency between the layers of the abstraction stack. This creates an opportunity, from the hardware architect’s perspective, to receive more feedback on the promoted innovations from all parties of the ecosystem.

5.4.2 Dual ecosystem

While we were eventually able to recognize the IT service industry as a particularly relevant candidate ecosystem, this recognition only clearly occurred in hindsight; during our work, typically within their organization; type II service providers are those providing IT services to multiple business units within their organization; type III providers cater to multiple external customers.
our choices were merely intuitively and subjectively motivated by past professional experience with this ecosystem. Instead, our immediate target was a project consortium named Apple-CORE, which we depict in fig. 5.4, and which shares design objectives (performance, efficiency) with the HPC community. In this ecosystem, four academic organizations play the roles of seven actors, identified as follows:

- a platform provider implementing the proposed architectural features in FPGA and emulation;
- an operating software provider for a machine interface language, extending C with primitives to control the new hardware features;
- another operating software provider for the SAC2C compiler, generating C code from Single-Assignment C (SAC) [GS06];
- another operating software provider for a parallelizing C compiler;
• three application providers for: existing benchmarks using plain C, SAC benchmarks, and hand-coded benchmarks using the proposed C language extensions.

Retrospectively, under the combined influence of both this ecosystem and the IT service industry identified earlier, our work has targeted a “middle ground” between the two. We acknowledge this as a shortcoming, because it prevented us from making choices that would have permitted performance improvements at the expense of portability, as would be appropriate in the HPC community, and choices that would have increased portability of the eventual platform to a larger diversity of application providers, as would be appropriate in the IT service industry.

Nevertheless, as we show in part III our choices were eventually successful at gaining early feedback, as per section 5.3, on the architectural innovation.

5.4.3 Ecosystem actors and their expectations

Our suggestion from section 5.4.1 was to target the ecosystem of IT service providers building upon F/OSS. In this community, the first level audience from the platform provider’s perspective is constituted by:

• F/OSS operating system providers (e.g. Solaris, GNU/Linux, BSD);
• F/OSS system-level language implementers (e.g. GNU Compiler Collection, LLVM);
• more recently, the providers of virtualization platforms, either encapsulated (e.g. Java, .NET) or para-virtualized (e.g. Xen, KVM, VMWare).

The diversity of actors in these categories has historically caused the community to self-organize around community-based standards that document their provided technology, in particular:

• codified ISO and IEEE specifications for the C and C++ languages [II11b, II11a] and the POSIX system interfaces [IEE08, II09];
• informal consensus about commonly supported extensions to these specifications; for example, the BSD network socket API and the GNU extensions to the C language.

With standards serving as middle ground with any downstream actors in the ecosystem, innovators cannot simply make deals with particular application providers to implement their own operating system and exploit their innovation directly; they must instead either provide one of the existing standard interfaces to their proposed technology, or argue to the community why the existing standard is inadequate and should be extended.

In the latter case, standardization of new interfaces first requires multiple proof-of-concept system implementations of the proposed interface by independent actors, then a proposal for a new standard, then community evaluation of the proposal. For example, this is the process that enabled the eventual integration of thread management in [II11b] compared to the previous [II99]. This process, while lengthy, promotes peer review and avoids tight vertical integration between hardware providers and specific application vendors.

In contrast, the lower interface between platform providers and the first level software actors is not codified and a large diversity of platform designs exists. This is a boon to hardware innovators, because it creates a tolerance for new platforms, and it pushes existing F/OSS operating systems to design for portability. This in turn significantly reduces the effort required to adapt an existing code base to a new hardware design.
CHAPTER 5. SYSTEM PERSPECTIVE

To summarize, the audience identified above expects the platform provider to interact with first level software actors towards providing a first implementation of the standard interfaces. It also expects the platform provider to simultaneously explain to third parties how they could exploit the platform on their own to deliver other implementations of the standard interfaces.

5.4.4 Strategy

In this context, a strategy to advertise the innovation from part I in this ecosystem would start with defining a hardware platform around the technology, then reuse existing F/OSS operating software components to provide (some of) the established standard interfaces to application developers, and then explain how this was done so that third parties could reproduce and/or extend the work.

We spell out these steps as follows:

1. describe the hardware/software machine interface;
2. define a hardware environment as will be observed by operating software components, including I/O devices;
3. implement a freestanding C language environment [II11b, 4§6, 5.1.2.1] which can compile autonomous code without external software dependencies;
4. port some operating system components using item 3 to provide hardware access to software;
5. port some C library components using items 3 and 4 to define a hosted C language environment [II11b, 4§6, 5.1.2.2];
6. document any unusual/new specific aspect of the platform required by third parties to reproduce steps 3 to 5;
7. illustrate with example/evaluation programs that exploit items 3 to 5.

The narrative of our dissertation essentially follows this plan. Part I, in particular chapter 4 covered item 1. We then address item 2 in section 5.5 below. Chapter 6 covers items 3 to 5. Chapters 7 to 11 address item 6. Item 7 is finally covered in part III.

5.5 Platform definition

To support the implementation steps sketched above, a platform with memories and I/O devices is needed around the processor chip.

The initial hardware environment implementing the architectural concepts from part I, available prior to our work, was extended as depicted in figs. 5.5 and 5.6. This equips the prototype microthreaded chip with an I/O system that can be connected to the necessary hardware devices, or low-level emulations thereof. More specifically, the FPGA integration was performed by another research group [DKK+12], and we extended the emulation environment to mimic the FPGA integration.

5.5.1 Support for I/O and legacy system services

Early on we observed that existing operating system code could not be reused directly with the proposed microthreaded architecture, even equipped with I/O devices. Indeed, all existing reusable operating system codes we could find, from embedded to server platforms
Figure 5.5: Extension of an FPGA prototype into a complete system.

This extension was performed by other researchers [DKK’12].

Figure 5.6: Extension of an emulation platform into a complete system.

This extension was performed as part of our work.
and from monolithic kernels to distributed systems, require support for *external control flow preemption* (traps and interrupts) at the pipeline level for scheduling, and either *IPIs* or *on-chip programmable packet networks* for inter-processor control and system-level communication. Meanwhile, the proposed design from chapters 3 and 4 omits preemption from the core pipeline to favor HMT instead. It also omits IPIs to favor a custom NoC supporting active messages [vECGS92] instead.

Because of this mismatch, any effort to port an existing operating system would require the system designer to either introduce the missing features into the architecture design, with the risk that these would impact performance negatively and add complexity to the machine interface, or redesign and re-implement the operating system components, which would add another significant development expenditure to the porting effort.

Instead, we took a transverse approach which we detail in [PLU+12]: add one or more *companion processors* implementing a legacy architecture to the platform, and use them to run one or more instances of an existing operating system. We call this *heterogeneous integration*. Once companion cores are available, it becomes possible to delegate any uses of system services from application code running on the new architecture to the companion processor(s).

The integration of a companion processor can be achieved either on-chip, i.e. on the same die, or off-chip, i.e. in a different chip package, in either of two ways:

- the companion processor shares the same memory system. In this case, the arguments and results of system services can be communicated implicitly through the cache coherence protocol. This is the approach taken in fig. 5.5b;
- the companion processor is a device on the I/O interconnect. In this case, the arguments and results of system services must be marshalled. This is the approach taken in fig. 5.6b.

The integration with shared memory provides the most flexibility for software and a higher bandwidth, but requires that the legacy core design can integrate with the memory network. Whether this is possible depends on the selected design: the address width, endianness and layout of data structures by the compiler must be compatible. In contrast, the integration on the I/O interconnect provides the most flexibility for the system integrator at the expense of increased access latencies and reduced bandwidth due to the need to marshal arguments and results.

Nevertheless, in either case, the porting effort for system services becomes minimal. The only new implementation required is a set of wrapper APIs on the microthreaded processors that serve as a *proxy* for a syscall interface implemented on the companion processor.

Although we developed this approach independently, we later recognized it in the “multikernel” approach proposed by the designers of Barrelfish [BBD’09], and at the larger scale in the use of AMD Opteron chips to serve as “service processors” in Cray’s XMT next to the main grid of simpler MTA cores [Kon07].

### 5.5.2 System architecture overview

From the software perspective, the platform is structured as follows:

- the microthreaded chip is connected to external memory and I/O devices through a system interconnect;
5.5. PLATFORM DEFINITION

- memory and I/O devices either share the same interconnect (e.g. fig. 5.5b), or memory uses a separate network (e.g. fig. 5.6b, cf. also section 3.4.1);
- a configurable Read-Only Memory (ROM) contains initialization routines and data structures that describe platform parameters; for example, the number of processors, the system topology, the access protocol on the NoC to the companion core(s), the location of devices in the address space, the cache sizes, the RAM size;
- there is at least one I/O device suitable to define an interactive console (e.g. a Universal Asynchronous Receiver-Transmitter (UART));
- there is at least one I/O device able to measure real time and trigger asynchronous events at specified times (e.g. a Real-Time Clock (RTC));
- there is at least one I/O device suitable to exchange larger data sets with the outside world (e.g. disk, network interface, etc.);
- there are one or more companion processor(s) running a legacy operating system for system services that cannot be implemented on the microthreaded cores;
- upon start-up, the microthreaded processor automatically starts a thread running the initialization code from ROM.

In the FPGA environment, the ROM contents are defined while configuring the fabric; in the emulation environment, the ROM contents are initialized from file in the host environment. Other I/O devices, like a graphical frame buffer and matrix displays were also implemented but will not be considered further here.

Summary

Avoiding the HIMCYFIO pitfall, and marketing architectural innovations into existing computing ecosystems require acknowledging both the current culture and assumptions of the candidate audience. The integration of new features into usable systems should avoid investing in entirely new operating software code bases and instead benefit from reusing the existing knowledge and software code base of the ecosystems.

- This in turn may constrain the hardware designer to provide some backward compatibility with past concepts. While this may feel like a restriction, it is also an opportunity to gain an early audience. Most importantly, any fundamental design issues will be revealed by legacy-oriented audiences just as well as they would be by custom, future-oriented new software stacks; meanwhile, they can be revealed earlier by targeting first existing ecosystems and taking their viewpoint.
- Applying these guidelines to the architecture from chapters 3 and 4, we have proposed in section 5.4 two candidate ecosystems and a strategy to achieve integration in this context. This in turn requires the definition of a hardware platform, which we provided in section 5.5, and allows us to plan software integration steps which will be followed by the following chapters.