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

Poss, R.C.

Citation for published version (APA):
Chapter 1

Introduction

Ingenuity and imagination, rather than accurate thought, are the ordinary weapons of science.

G.H. Hardy [Har11].

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Birdview epistemology</td>
<td>16</td>
</tr>
<tr>
<td>1.2</td>
<td>Computer science</td>
<td>18</td>
</tr>
<tr>
<td>1.3</td>
<td>Stem cells of computing</td>
<td>20</td>
</tr>
<tr>
<td>1.4</td>
<td>Losing sight of generality</td>
<td>24</td>
</tr>
<tr>
<td>1.5</td>
<td>Achilles’ heel of designers</td>
<td>26</td>
</tr>
<tr>
<td>1.6</td>
<td>Case study: hardware microthreading</td>
<td>28</td>
</tr>
<tr>
<td>1.7</td>
<td>Overview</td>
<td>28</td>
</tr>
</tbody>
</table>
1.1 Epistemology: science is team work, so is innovation

The traditional purpose of the fundamental sciences is the acquisition of new knowledge pertaining to observed phenomena, in an attempt to describe “what is.” In parallel to the discovery of new knowledge through scientific inquiry, philosophers, or theoreticians, derive ideas of “what could be.” Via formalisms, they construct structures of thought to validate these ideas and derive iteratively new ideas from them.

We can focus for a moment on the human dynamics around these activities. On the one hand, the intellectual pleasure that internally motivates the human scientists is mostly to be found in the acquisition of knowledge and ideas. For natural scientists, the focus is on accuracy relative to the observed phenomena, whereas for philosophers the focus is on consistency. On the other hand, the external motivation for all fields of science, which materially sustains their activities, is the need of humans for either discovery or material benefits to their physical existence. From this position, the outcome of scientific inquiry and philosophical thought, namely knowledge and ideas, is not directly what human audiences are interested in. The “missing link” between scientific insight and its practical benefits is innovation, an engineering process in two steps.

The first step of innovation is foundational engineering: the creative, nearly artistic process where humans find a new way to assemble parts into a more complex artifact, following the inspiration and foreshadowing of their past study of knowledge and ideas, and guided by human-centered concerns. Foundational engineering, as an activity, consumes refined matter from the physical world and produces new more complex things, usually tools and machines, whose function and behavior are intricate, emergent composition of their parts. The novelty factor is key: the outcome must have characteristics yet unseen to qualify as foundational; merely reproducing the object would just qualify as manufacturing. The characteristic human factor in this foundational step is creativity, which corresponds to the serendipitously successful, mostly irrationally motivated selection of ideas, knowledge and material components in a way that only reveals itself as useful, and thus can only be justified, a posteriori.

The other step is applicative engineering, where humans assemble artifacts previously engineered into complex systems that satisfy the needs of fellow humans. In contrast to foundational engineering, the characteristic human factor here is meticulousness in the realization and scrupulousness in recognizing and following an audience’s expectations—if not fabricating them on the spot.

The entire system of activities around science is driven by a demand for applications: the need of mankind to improve its condition creates demand for man-made systems that solve its problems, which in turn creates demand for new sorts of devices and artifacts to construct these systems, which in turn creates demand for basic materials as input, on the one hand, and intellectual diversity and background in the form of knowledge and ideas. We illustrate this general view in fig. 1.1, and argue it is also valid for fundamental sciences in side note 1.1. The role of education, in turn, is to act as a glue, ensuring that the output of the various activities are duly and faithfully communicated to the interested parties.

Our first observation, which is perhaps the key motivation for the work presented in this dissertation, is that different humans who partake in these activities have different preferences. We do not expect any one person to participate in and be successful at all steps to improving the condition of mankind. The corollary is that for all processes to be successful, humans must acknowledge their separate interests and coordinate their work towards the common goals.
1.1. BIRDVIEW EPISTEMOLOGY

---

**Figure 1.1: Activities related to science.**
Side note 1.1: Fundamental sciences are also application-driven.

One may argue that the search for knowledge and understanding can be an end in itself, and deserve support (e.g. via funding) regardless of its applicability. However, we argue that all scientific activities are motivated by human needs for either discovery or comfort, and that it is human demand for their utilitarian value that fuels continued inquiry.

The need for discovery concerns the expansion of mankind, both over time, space and intellectually. Discovery has been historically more valued; it has fueled e.g. investigation into navigation and astronomy to conquer space, history and medicine to conquer time, mathematics, logic and linguistics to conquer the mind, theology, politicoology and sociology to conquer the human group. Even astrophysics, quantum physics and cosmology can be seen as discovery means towards conquering the universe. All other fields of science exist to either improve the scientific process towards discovery, e.g. artificial intelligence or computational sciences, or to optimize the use of the already-conquered area by humans, i.e. increase comfort.

In either case, the outcome of the scientific activity must be engineered into useful objects before it becomes visible, even when the objects are just knowledge vehicles like books or lectures. Creating a book, for example, involves a foundational step in expressing the text by the author, then assembly of the text into a concrete print by the publisher. Even with “fundamental sciences” which claim to produce only abstract knowledge, the value of the scientific activity can only be ascertained once the acquired knowledge is actually communicated. We can thus say that it is the demand for knowledge vehicles, expected as physical artifacts, that fuels the fundamental sciences.

1.2 Computer science, engineering, and its architecture sub-field

The term “computer science” broadly encompasses all activities around the design, manufacturing, application and analysis of computing systems. Computer systems engineering is its subset of activities related to the design of the computing artifacts, i.e. the design of tools that can be subsequently applied by the IT industry to mankind’s interests. We illustrate the further sub-fields of computer engineering in fig. 1.2, namely: electrical engineering, software engineering, computer architecture and systems architecture. As highlighted above, the foundational part of computer systems engineering is inspired by, and based on knowledge and ideas from natural sciences, such as biology and physics, and philosophy, such as the theoretical work of George Boole, Alan Turing, Alonzo Church and Claude Shannon; its outcome, i.e. computing systems, is then instantiated in computing applications, such as home appliances, cars or medical equipment.
In contrast to some fields of technology that can distribute the various activities to separate groups of people, creating a divide of social roles between “scientists,” assumed to be out of touch with the needs of daily life, and “engineers,” assumed to be out of touch with fundamental theory, computer science is peculiar in that it requires all its practitioners to be both skilled engineers and competent fundamental and experimental scientists. This is a characteristic shared by all fields where the produced systems are complex assemblies of parts themselves complex, deployed in complex situations; avionics, high-speed trains and spatial equipment are other such fields. The reason is that for the composition of complex pieces into a complex system to be tractable by human practitioners, first a model of the parts’ behavior must be devised, to simplify the mental image manipulated in the creative process in the next step. For example, when computer architects assemble caches and processors into a multi-core microprocessor chip, they do not keep track of the individual composition of the components in terms of p-type and n-type transistors; instead they manipulate an idealized model of their behavior derived from previous implementations by the experimental and analytic scientific inquiry. This abstraction process happens at all levels of composition, from the grouping of silicon crystals into semiconductors to the grouping of individual processing units into networks of computing clusters.

Historically, computer science was first an isolated field whose end applications were restricted to defense and high-profile business administration. States (for defense) and large companies (for business) could fund single organizations with large research facilities to design their systems. When computers became useful to other applications with more democratic audiences, the audiences could no longer directly fund single, large organizations. The design of computing tools became distributed across distinct organizations carrying the innovation process at different levels, and interacting together to combine systems. A key enabling factor for this division of concerns was the invention of the general-purpose computer, which allowed to separate the designers of hardware from the designers of software into distinct communities.

1.2.1 The wonder and promise of general-purpose computers

In the middle of the 20th century, something exceptional in the history of mankind happened: a universal tool was invented: the general-purpose computer. For the first time, any human could potentially become an inventor and solve arbitrarily complex problems using pictures of their thought processes projected as information patterns, i.e. programs, without the cost of manufacturing new physical artifacts.

Although Charles Babbage can be acknowledged as the forefather of this artifact [Hal70], the full scope of its generality was only understood by Turing at the start of the 20th century. Indeed, our current general-purpose computers approximating Turing’s abstract machine are one of only two ways that we currently know to make a computer that can compute most of what a human may want to compute; the other way being queue machines, invented later [FE81]. Furthermore, when the computer is connected to input/output interfaces, it becomes interactive and can convert to and from phenomena of the physical world. Conversely, there is no known other way to assemble artifacts, other than those that can be described by interactive Turing and queue machines, which can compute most of what humans can think of in a way that can influence, or can be influenced by, physical phenomena.

Obviously one does not need to invoke Alan Turing nor his thoughts when building a device that accomplishes a specific task. Census count machines reading punched cards were built and used successfully long before Turing was born [Hol89, Ran82]. The reason
why Turing’s contribution was remarkable is that it created theoretical confidence that a
general-purpose hardware platform could be successfully reused, after it is fabricated, to
solve problems not defined yet, and thus guaranteeing perpetual employment for software
creators.

This confidence stems from the formal thesis of Alonzo Church and Alan Turing on
computability—although it really comes from collective work with Post, Kleene and Gödel,
cf. [Dav65] for the full story—which establishes that all expressible formulas of arithmetic,
which by definition are all possible computations that humans can phrase intelligibly ever\(^1\),
can be computed by either an abstract Turing machine, Church’s \(\lambda\)-calculus or partial re-
cursive mathematical functions\(^2\). Moreover, when the machine is universal, the function it
computes can become a run-time input, i.e. software, while preserving the full generality of
the model. Because of this, a hardware platform that resembles a universal Turing machine
gives us confidence that it can be reused in the future by new software to solve problems
that have not been formulated yet. Since the only conceptual difference between a uni-
versal Turing machine and a concrete implementation is the finiteness of storage capacity
(vs. the infinite length of Turing’s tape), it is possible to approximate the abstract machine
increasingly accurately by simply adding more addresses to the storage, which seems to be
technically tractable for the foreseeable future.

This is the crux of what general-purpose computing is about: design and build a hardware
platform now, with reasonable confidence founded in logic that they can be used to solve
future problems in software.

Since then, other formalisms distinct from Turing machines and \(\lambda\)-calculus have been
developed and subsequently proven to be Turing complete, that is, at least as powerful as
Turing machines. The models that have received most attention are queue machines
(mentioned above), graph reduction machines able to reduce terms of \(\lambda\)-calculus [CGMN80,
PJCSH87], register-based variations on the Turing machine [vEB90], the \(\pi\)-calculus [Mil90],
and specific initial states of cellular automata [Cha02, Coo04]. Any of these models can
be made universal, i.e. programmable via software, by modeling a single-function machine
whose behavior is to read a program from an input device and then evaluate it. They
can furthermore be made interactive, i.e. connected to the physical world, by introducing
basic actions in their evaluation rules that effect external interactions. However, the only
computing artifacts built so far have been either Turing-like (register machines) or queue-
like (stack machines). All implementations of other formally as powerful models have only
existed as simulations of their semantics in programs running on hardware devices that can
be primarily described as register or stack machines. It is the Turing-completeness of these
specific two models that guarantees the future utility of the constructed artifacts.

### 1.3 General-purpose computers are the stem cells of computing

Sometime between 1992 and 1996, CALC was written. CALC was a graphing program:
the user would interactively enter on the keyboard the definition of a function and the
coordinates of a view window, and the program would plot the function, one point per

---

\(^1\)NB: computations (computable functions) are a subset of all functions that can be phrased by hu-
mans. In particular there exist intelligibly phraseable non-computable functions, such as the busy beaver
function [Rad62].

\(^2\)It further establishes that neither of these formalisms can compute anything more, that is, everything
we can compute using either can be described by an intelligible, valid formula of arithmetic; but this point
is irrelevant here. See [Hof99] for an accessible review of the theoretical and practical consequences.
column of the graphical display. As the story goes, CALC was written in BASIC over the course of several months; a few months afterwards, the only extant copy of CALC was lost.

We resurrect the memory of CALC here to highlight the role of general-purpose computing. Indeed, CALC would allow the user to enter any function definition that was valid in BASIC. The syntax allowed integer and floating point arithmetic, grouping parentheses, operator precedence, and uses of any built-in functions. It would then plot that function interactively, i.e. without having to stop and re-run the program. In other words, the program would understand a phrase expressed in a human language, that of mathematics, and act upon it automatically. Yet, implementing that feature was trivial: CALC would simply write the text of the user-supplied expression into a file, and load back the file into the BASIC interpreter as an additional program fragment\textsuperscript{3}.

To understand how this is relevant here, one needs to consider this anecdote as a parable. What happened really is that an uneducated person was empowered to create by a programming environment which was, through its naive simplicity and despite its flaws, intendedly devoid of any specific purpose. A simple general feature, namely the ability to read a user-defined program text from input and evaluate it, was key to overcoming the most complex theoretical aspect of the task at hand. This parable illustrates that general-purpose computing platforms are, like the stem cells of living organisms, key to the perpetuation of computer engineering. They empower practitioners, both amateur and seasoned, to express their creativity past the bounds suggested by current applications and uses, and solve new problems in revolutionary ways.

There are two reasons why this generality is strongly desirable. The first reason is that innovation and major advances in the field are a creative process by humans for humans, as highlighted above. Creativity in humans usually occurs only in unbounded conceptual frameworks and playgrounds. Therefore, computing science, as a field, will need flexible and generic platforms for new developments and innovation. These platforms might be isolated, conceptually or physically, from the products available to the general public, but even when so pressured they will continue to exist as an essential and durable niche market for computer science practitioners themselves.

The second reason is that all current trends converge towards the second era of separated computing\textsuperscript{4}, with visible and much-awaited benefits in terms of energy and cost management.

The visible tip of this iceberg, on the network side, is perhaps the ongoing rise of social networks and online sharing platforms. But even in corporate environments, more and more responsibility, in particular regarding the safeguarding and consolidation of data, is pushed away from workstations to networked locations and accessed remotely. This setup principally enables sharing the infrastructure costs (security, cooling, storage, failure management) for the compute-intensive parts of networked applications. It reduces synchronization and communication latencies in large applications by literally increasing locality, namely by grouping the communication-intensive parts into a close geographical location. Through careful over-subscription of shared computers, it also distributes the energy investment more equally across heterogeneous applications. This setup is technically usable nowadays, as opposed to the last part of the previous century when the client-server model somewhat waned, essentially because of lower latencies in networks (cf. side note 1.2).

Meanwhile, and perhaps paradoxically, the devices at the human-computer interface become increasingly powerful. Current low-end gaming devices already offer full virtual immer-

\textsuperscript{3}Using the \texttt{CHAIN MERGE} statement.

\textsuperscript{4}The words “separated computing” include both the asymmetric client-server model and distributed applications where the overall behavior emerges from equal contributors.
sion through rich auto-stereoscopic images [JMW`03, Tab10, Lea10]. Reality-augmenting
glasses with on-demand, real-time streaming of contextual data are on the verge of becoming
mainstream [HBT07, Ber09]. All futuristic visions of human-centric computing include pervasive and seamless human-computer interaction with incredible (by today’s standards) amounts of signal processing.

To maintain control on power usage and locality, most of the signal processing will need to be physically performed at the site of perception. What we currently call high-performance computing equipment will find its way to the wearable miniature sensors of our future selves. However, for no less obvious reasons, the processed data will flow between the individual and the collective self, through distributed networked applications, because only there can the sense-data receive the meaning necessary to its processing\(^5\).

Without speculating further on the nature of these dual computing systems made of intelligent sensors and networked applications, it seems reasonable to assume they will be based on hardware components responsible for transforming information. These future systems may bear little resemblance to our current technology; yet, regardless of their exact nature, one of their characteristics seems inevitable: adaptability.

Adaptability is the feature that will support technological evolution under the selective pressure of market effects. Indeed, unless disaster strikes and totalitarian regimes become the norm, free exchange of ideas and objects will force a dynamic, fast-paced adaptation of technology to evolving human interests. Even assuming a stabilization of human demographics, the increasing access to technology and networks will cause the market for computing systems to become even more segmented than today, with entire verticals\(^6\) rising and falling faster than education systems. Combined with the fact that the knowledge required to comprehend and maintain systems will be increasingly dense, and thus decreasingly accessible, there will not be enough manpower to design and implement entire new systems to cater for new verticals. Since there is not yet any confidence that new designs can be reached via autonomous artificial intelligence, we should assume instead that guided adaptation of existing concepts to new uses and new requirements by humans will be the norm.

Evolutionary theory suggests that adaptation works best if the system keeps a healthy reserve of repurposable stem cells. It seems conceptually difficult to re-purpose the programmable controller for a washing machine into a car navigation system; whereas the computer scientist today clearly sees a specialization path from a general-purpose computer to both devices. Actually, specialization of computing elements, like cell differentiation in organisms, is an unavoidable phenomenon required to support the increased complexity of their applications. However efficient specialization is a repeating phenomenon, with each generation stemming from non-specialized components instead of previous generations of specialized systems. This applies to both hardware design and software design.

In the light of this perspective, one could possibly accept the doom of commodity, one-size-fits-all “all-purpose” computer designs. Individual devices that would truly satisfy any computing need in a practical or economical way have never really existed. Besides, the immediate human-computer interface is best served by specialized devices. However, general-purpose specializable computing systems must continue to exist, at least for those humans who, through their creativity and inventiveness, will be responsible for future innovation.

\(^5\) This vision of networked computing was inspired by professor Zhiwei Xu, from the Chinese Academy of Science.
\(^6\) Vertical: an entire computing market addressing a specific segment, from hardware manufacturing to online services including system and software development. Two archetypal verticals are mobile phones and car navigation systems.
1.3. STEM CELLS OF COMPUTING

### Side note 1.2: The second rise of separated computing.

Historians of our field have highlighted that client-server computing has been prevalent before, but then disappeared. This warrants an aside.

The key observation is that the client-server model “makes sense” economically and practically for the reasons mentioned in section 1.3. These merits have never disappeared; they have been merely shadowed by the ever-increasing computing speed of autonomous, commodity general-purpose computers (“desktops”) in the last period of the 20th century relative to the network latencies.

Indeed, the centralized structure of the first networks, combined with the characteristics of network links at that time, was a serious obstacle to their increased prevalence. To match the increasing expectations of computer users, applications had to become more interactive and more responsive. More than the load on computational performance on servers, the subjectively unacceptable response time and jitter of communication links was the first motivation towards increasingly powerful terminals and locally interactive applications. Previous work on human-computer interaction [Mil68, CMN83, CRM91] suggests that 100ms is the maximum acceptable delay between a keystroke and the screen response, and [Shn84] suggests that higher variability of response times incur lowered user performance. These findings have served as references for user interface design ever since [Nie94, Chap. 5]. In contrast, the high latency and contention of terminal channels to shared computing facilities in the period 1970-1990 was failing users on both counts.

The market for personal computing grew initially faster than communication networks, further strengthening the trend toward autonomous computation devices, incarnated in personal computers. The low-latency, high throughput, globally available distributed network that was needed to support the growth of a computing market based on the client-server model only appeared much later—and unfortunately long after audiences had gotten used to the merits and flaws of their autonomous, power-hungry, inefficient commodity general-purpose computers. In contrast, the reason why the Internet became what it is today was a combination of economic and social factors outside of the realm of computer science: the global need for more connectivity, more trade, information exchange, etc. The epic conflation of telephone and data networks is fueling, to this day, a fierce competition that produces as a side effect an increasingly large, robust and wider network.

Now that the global network is available, we should find it remarkable that all online actors, from individual news broadcasters to scientific computing centers, know acutely the huge potential of such a strong connectivity. It is just natural that we use this opportunity to again tap into the merits of separated computing, which we had left aside for a few decades.

---

### Side note 1.3: Embedded/specialized vs. general-purpose computers.

A defense of general-purpose computing would be incomplete without a reference to embedded systems. The increasing pervasiveness of embedded systems is often cited as a tendency towards the disappearance of general-purpose computing. However, this is likely a fallacy.

The characteristic of embedded systems that is relevant here is their invisibility: while they are necessary to technological evolution, and their function is expected by the users of the devices where they are embedded, these systems are not shown. In fact, embedded systems do not have an existence as computers in the eye of their users, which therefore do not expect from them “abstract” features such as re-programmability and re-configurability. The large and growing embedded market is one of specialized function, reliability, sometime performance, and foremost dedication to well-defined tasks that allow economies of scale in manufacturing. Embedded devices are essentially engineered for specific purposes at the expense of programmability and configurability, which typically matter less to their audience.

When considering their manufacturing and sales figures, and the massive body of knowledge developed to support their field, embedded systems dominate the field of computer science by sheer numeric superiority. It is this pervasiveness that seems to shadow general-purpose computing and suggests its doom from the perspective of practitioners in the field.

However using the growth of embedded systems as an argument against general-purpose computers amounts to ignoring the “elephant in the room”: the creation of embedded devices would not be possible without development computers, those very computers used by embedded system engineers: developers, testers, quality controllers, etc. By the initial argument above, most of the advances in embedded system design would not be possible without *stem computers* that let the designer imagine radically new devices without preconceptions: either their own development machines, or non-specialized, freely composable template components with no predetermined function.
Chapter 1. Introduction

Side note 1.4: Example functions that can be made primitive recursive.

Multiplying two matrices; filtering noise out of audio; compiling assembly code to machine code; computing a minimum flow in an acyclic graph; recognizing object boundaries in an image; plotting a graph for a fixed arithmetic function; printing a JPEG image; looking up a value in a dictionary; simulating another computer which can compute only primitive recursive functions.

Side note 1.5: Example functions that are not primitive recursive.

Sorting; recognizing a tune from an audio sample; compiling a C++ program to machine code; computing a shortest path in a cyclic graph; recognizing objects in an image; plotting a graph for an arithmetic function entered interactively; typesetting a TeX document; running an SQL query on a relational database; simulating another computer which can compute partial recursive functions.

1.4 Losing sight of generality: the risk of throwing out the baby with the bathwater

Half a century after having found a true wonder, we run the risk of losing it: under the pressure of capitalism, optimization and efficiency, the generality of computers has been endangered since the turn of the 21st century.

The prevailing thought in the personal computing industry at the time of this writing is that there are no uses of computing systems by the general public that are freely programmable by the user\textsuperscript{7}. A common argument in favor of such an iWorld, where form is actualized by branded functions, and where each fixed-function iApplication is blessed individually by arbitrary and opaque corporate review processes—or euthanized before birth if its market value is not ascertained upon inception—is the growing enthusiasm for iThis and iThat devices initiated with the new millennium.

The forces driving this evolution are further discussed in [Zit08]: the free market and deregulation of large companies has created incentives to innovate behind closed doors and create instead devices that capture their users into consumption cycles. Capital gains discourage reuse and extension of existing products in favor of the forced acquisition of new products for every new application. The associated risk is an opportunity loss: individuals with the skills to innovate in software become increasingly limited in their ability to carry out innovation until they are trained privately by corporate technology providers.

Even in technical circles, where generality is traditionally respected, the need for increased computing power and efficiency creates pressure to reduce generality. This effect has existed throughout the history of computing; for example, the author of [Day11] reminds us how Dijkstra’s suggestion to introduce recursion in programming languages was met with resistance from audiences focused on the short-term performance gains from non-recursive language semantics. Yet, for reasons we will revisit below in section 1.4.1, the computer engineering field has come lately under tremendous pressure to answer requests for additional performance, and the temptation is great to specialize devices to match this demand.

We can recognize this situation in the recent enthusiasm for “accelerator” boards, which re-purpose graphics-oriented co-processors (usually texture shaders) towards other types of computations. The marketing message is that these processors are “general-purpose”: they seem to address application needs of both scientific and consumer audiences alike. However this message is misleading: full generality requires interaction, arbitrarily large

\textsuperscript{7}Including, but not limited to, the ability to share freely the outcome of a programming activity with peers.
random-access storage and either arbitrary numbers of conditional branches or arbitrarily deep data-dependent recursions, *per individual thread*, which these devices usually do not support\(^8\). Instead, they are just sufficient to compute any *primitive recursive function* of arithmetic\(^9\). Since most of what humans need to compute can be described by primitive recursive functions (cf. side notes 1.4 and 1.5), this type of computer seems already quite useful; however, by the argument of section 1.2.1, these devices fundamentally limit our ability to solve future problems.

### 1.4.1 The current pressure to innovate

Processors and memories are two unavoidable sub-parts of any computing system, as they are at the heart of the ability of the system to compute (cf. section 1.2.1). Until the turn of the 21st century, system engineers using these components as building blocks could assume ever-increasing performance gains, by just substituting any of these components by the next generation of the same. Then they ran into two obstacles. One was the *memory wall* [WM95], i.e. the increasing divergence between the access time to memory and the execution time of single instructions. The second is the *sequential performance wall* [AHKB00, RML\(^0\)01], i.e. the increasing divergence in single processors between performance gains by architectural optimizations and the power-area cost of these optimizations. A third potential wall is now visible on the horizon: the end of Moore’s law [Kis02, ZCHB03, TP06], or more precisely a potential limit on the maximum number of silicon CMOS-based transistors per unit of area on chip. These obstacles are the stretch marks of a speculation bubble ready to burst. Unless solutions are devised within a decade, the economy of the IT industry of applications, currently based on an expectation of future cheap performance increases in computing systems, may need a serious, globally disruptive reform.

### 1.4.2 Where and how to innovate

The discussion so far opens space for innovation in different directions. A first possible direction is to invent a new implementable universal computing model that offers at least as much expressivity as Turing’s model, with more computing power and scaling opportunities than what all current devices offer. Quantum or biological computers may be candidates, although their realizability at the scale of current human needs, and their universality, are not yet ascertained. Another direction is to find new smaller and faster building blocks to make universal and interactive Turing-complete computers, for example using light-based logic instead of electronics [SLJ\(^+\)04], however these new technologies may not be available before the limits of silicon scaling are reached. In the medium term (a decade), innovation seems tractable at two levels. Software ecosystems could embark on an intensive quest for simplification towards fighting Wirth’s law\(^10\) and gaining efficiency. However, the odds that this will happen are at best unclear [Ken08, MSS10, XMA\(^+\)10]. Or, *computer architects* could find new ways to arrange CMOS logic into different, more efficient combinations of processors and memories, possibly exploiting parallelism and scalable throughput. This direction, initially suggested in [RML\(^0\)01], seems especially tractable given the growing availability of concurrency in software, a topic we revisit in chapter 2.

---

\(^8\)We support this argument for an example state-of-the-art accelerator chip in chapter 12.

\(^9\)To simplify, a primitive recursive function is a computation where the number of steps in any repetition is known before the repetition starts; for details see [Pét34].

\(^10\)“Software is getting slower more rapidly than hardware becomes faster.” [Wir95]
CHAPTER 1. INTRODUCTION

1.4.3 Politics and openness in innovation

Assuming computer architects will be responsible for principal innovations in computer engineering in the next decade, and assuming generality in the produced systems is a desirable feature for software audiences, there exist two principal questions:

- the *inner question*, from the individual architect’s perspective, is one of substance: what will be the good ideas and new component assemblies? What will they look like?
- the *outer question*, from the perspective of the computing science community, is one of logistics: how to ensure that any innovation, if it occurs, will be brought to realization and support the continued growth of computing ecosystems?

Answering the inner question is fully under the responsibility of the architects, and bounded only by their creativity. The outer question, however, is not under their control. As we identified in section 1.4, market effects combine the overall need for new technology with an incentive for corporations to keep their solutions specialized and opaque. Levers to market dominance and corporate power struggles, in particular increasing uses of the vendor lock in and planned obsolescence effects, create the risk that any coming innovation in computer architecture will be realized behind closed doors, and will only be featured in products lacking the desired generality.

The computer architecture community is responsible for safeguarding against this appropriation of upcoming innovations by corporate interests. The methodology to guarantee the generality of solutions and their accessibility to large audiences is obviously transparency, that is documenting publicly not only the outcome of the innovative processes but also which creative steps were undertaken and their background knowledge and experiences. This is necessary so that other parties, in particular younger generations of computer engineers, can steadily join the innovation enterprise via imitation, reproduction and extension of past discoveries.

1.5 The Achilles’ heel of designers: HIMCYFIO

On the road to innovation, computer architects face a pitfall which we call “Here Is My Component, You Figure It Out (HIMCYFIO).”

The circumstance for HIMCYFIO concerns visionary computer architects who address a known or future technology obstacle while avoiding complexity, i.e. who deviate from established routes while preserving conceptual simplicity. The pitfall occurs when the inventor proposes a new component design, in isolation, with the unfounded confidence that its purpose and utility are both self-explanatory and desirable, and thus that it will be “necessarily” successful upon completion. The assumption is that industry will “eventually catch up” and integrate the component in larger computer systems, because the need seems self-explanatory (cf. section 1.4.1) and the integration self-evident (seems simple to the designer himself). This assumption is fallacious because peer inventors and potential users, especially the software community, will both find the invention foreign, i.e. difficult to understand simply because it differs from the common ground, and non-trivially applicable, because its proper use in larger computer systems is not yet determined.

To summarize, HIMCYFIO occurs because hardware architects mistakenly consider that the novelty and simplicity of a new approach is sufficient to relieve them of the burden of holding their audience’s hand in appreciating their work. In even shorter words, HIMCYFIO occurs when architects answer their inner question while avoiding or answering incorrectly
the outer question (cf. section 1.4.3). When HIMCYFIO occurs, potentially interesting component designs run the risk of never being integrated in a working computer. Large research investments, including potential great ideas, may be lost; personal ambitions may be crushed. We could find several example victims of HIMCYFIO:

- the Manchester dataflow machine [GWG80] proposed a relatively simple processor design using dataflow principles, with the early vision that the latency of all external communication, even fine-grained memory accesses, should be tolerated by selecting independent instructions in hardware. However its exclusive dedication to SISAL [MSA+83] forced its designers into a niche community where they stayed until the termination of the project in 1995;

- the designers of the Monsoon architecture [PC90] had a similar combination of fresh applications of dataflow scheduling and practical implementation plans for a processor. However they restricted the use of their design to the Id language [ADNP88], while suggesting that I/O and program control would be handled on a separate type of processor running the (by-then-already-standard) Unix operating system. This forced users to work with two programming models, and the resulting conceptual conceptual complexity proved fatal to the project;

- a more recent and high-profile example can be found with Transmeta’s Crusoe architecture [Kla00]. On the one hand, the combination of a relatively simple Very Large Instruction Word (VLIW) design with Code Morphing [DGB+03] was a revolutionary way to expose a high-ILP, low-power pipeline to an audience locked into Intel cores. On the other hand, the final product’s packaging was ill-designed: despite the presence of an on-die DDR memory controller (for the instruction cache), external memory was supported only via a slower SDRAM interface; and I/O was restricted to the legacy “southbridge” system interface. These integration choices rendered the overall chip un-competitive for data-intensive or graphics-intensive applications. While Transmeta’s goal was to guarantee thermal budget envelopes, and the chip was retrospectively competitive in this regard, this priority was not appropriately communicated to the company’s audience.

These examples illustrate that to avoid this pitfall, architects should design new components in the context of entire systems and their applications:

- when designing for new applications, they must work together with system and compiler programmers so that this first level community obtains an early understanding of the architecture’s benefits;

- when designing for existing applications, they must acknowledge the assumptions made in the existing programs and provide designs that integrate existing usages transparently; in particular, they must acknowledge common expectations about performance without requiring radically new machine models.

The reason why recognizing and averting HIMCYFIO systematically is important is that this is the only way to make generative architecture research an attractive field for newcomers.
1.6 Case study: hardware microthreading

In an effort to innovate in computer architecture while preserving generality, as identified in section 1.4, a research project coordinated from the University of Amsterdam is exploring new ways to assemble microprocessors from logic. After observing that a large amount of untapped concurrency is already present in software applications, the researchers in this group propose to design processor chips that cater primarily to throughput scalability and execution efficiency, possibly at the expense of the performance of single instruction streams. The proposed chip design combines short Reduced Instruction Set Computer (RISC) pipelines with dynamic dataflow scheduling [NA89] and Hardware Multi-Threading (HMT) on individual cores for latency tolerance and energy efficiency, and implement flexible hardware support for thread and task management across different cores for throughput scalability. We shall call this approach “hardware microthreading,” as opposed to simply “microthreading” used in prior work from this research team, to distinguish it from the microthreads managed in software in IBM’s Cell Broadband Engine [AAR08].

The abstract definition of hardware microthreading, together with guidelines on how to design architectures around it, establishes principles of architecture design. These should in turn be applicable to a diversity of application domains, from embedded systems to high-performance supercomputers. However, before this happens the principles must be illustrated at a small scale, into an artifact suitable to convince external observers, as highlighted in section 1.5.

To achieve this, the research group also attempts to define an entire general-purpose chip architecture which embodies hardware microthreading with simple RISC cores and a custom Network-on-Chip (NoC). In the preliminary phase, given the complexity of modern silicon, this research was carried out via detailed simulations of individual components based on their potential behavior on silicon, using artificial microbenchmarks. The initial results were encouraging [BHJ06a, BHJ06b, BGJL08], and motivated the enterprise of a larger project to carry out a more extensive realization:

- at the hardware level, design and implement a prototype single-core implementation of the new architecture on an Field-Programmable Gate Array (FPGA); and simultaneously implement a software emulation of a full multi-core system using the new processor design;
- at the software level, design and implement new operating systems, language tool chains, and a representative set of benchmarks to evaluate the new hardware architecture, both on single core (FPGA prototype) and multi-core systems (emulation environment).

This is the project where this book originated.

1.7 Overview

Our dissertation reflects on the interaction between foundational engineering activities around hardware microthreading. It provides two reading levels.

At the first level, a technical report provides possible concrete answers to the outer question around hardware microthreading. This report recollects the substance of the invention in part I, then describes in part II how we extended the base architecture concepts into a
Table 1.1: Symbols used to mark technical contributions throughout our dissertation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>○</td>
<td>Exposition of previous answers to the inner question.</td>
</tr>
<tr>
<td>□</td>
<td>Argument or contribution towards answering the outer question.</td>
</tr>
<tr>
<td>■</td>
<td>Contribution towards answering the inner question, required while answering the outer question.</td>
</tr>
<tr>
<td>▷</td>
<td>Acknowledged opportunity for further work on the outer question.</td>
</tr>
<tr>
<td>▶</td>
<td>Required further work on the inner question, identified while answering the outer question.</td>
</tr>
</tbody>
</table>

general-purpose platform, and finally summarizes key evaluation activities around this platform in part III. During this research, we also made related minor contributions to the substance of the invention. We classify our findings about the inner and outer questions around hardware microthreading throughout the text using the symbols from table 1.1; a summary is given in table 1.2. This technical contribution touches multiple system-wide issues, from code generation to applications through the operating system stack; as such, it connects architecture and system research [MBRS11]. The scope of the work at this reading level is illustrated as the light gray area in fig. 1.3: this dissertation complements [Lan07, Lan1x], which mostly describe the answers to the inner question (dark gray in the figure); it also does not consider how the innovation will eventually be used in specific applications.

The main contributions are to be found at the second reading level. Our choice of hardware microthreading is really a case study for the dialogue between the innovator and their community of peers. At this level, we make two contributions. The first contribution is a methodology that a chip processor designer can use to avoid the HIMCYFIO pitfall:

1. exposing publicly the design concepts and their trade-offs (part I),
2. recognizing and choosing audiences, and using their current expectations as a starting point to define a context (chapter 5);
3. explaining the innovation to the chosen audience(s) in the way that the audience was previously used to (chapters 6 and 7);
4. exposing and detailing how the innovation differs from past experience of the audience, within the context determined in step 2 (chapters 8 to 11);
5. providing a test and experimentation environment to third parties, and letting the chosen audience observe that third parties can use the innovation on their own (part III).

This methodology is presented throughout the dissertation, and applied to the topic of hardware microthreading as an example.

The second contribution is a discussion, spread over the entire dissertation, about what “generality” means and what features are required in computing components so that they can be advertised as general. As we argued previously in section 1.3, the future of automated computing will require humans to specialize “stem cell” computers towards specific applications. Within the context of computer architecture, the properties required from these “stem cells” have been outlined abstractly in section 1.2.1; the discussion in part II shows how to find or construct these properties in concrete implementations.
Figure 1.3: Scope of the answers to the outer question.

The dark gray area highlights the scope of [Lan07, Lan1x].
The light gray area highlights the scope of this dissertation.

Table 1.2: Overview of technical contributions per chapter.