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

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 12

Issues of generality

Abstract

In section 1.4.3, we have highlighted that innovation in computer architecture is composed of two parts, one that answers the “inner question” by designing the substance of the innovation, and another that answers the “outer question” which places the innovation into its context for consideration by external observers. In [Lan07, Lan1x], the author answers the inner question for hardware microthreading in depth, by detailing the hardware design and intrinsically demonstrating its behavior. As part of our process to answer the outer question, we have exposed so far the features of the design visible from software, which we will illustrate in the context of existing software ecosystems in part III. There is however one aspect not covered by the previous chapters of this dissertation nor [Lan07, Lan1x]: the argument to the audience about the generality of the invention. In this chapter, we argue that the proposed design provides fully general cores and is suitable to support most parallel programming patterns.

Contents

12.1 Introduction ................................................................. 188
12.2 Turing-completeness: one and also many ................................ 188
12.3 Abstract concurrency models ........................................... 189
12.4 Turing completeness, take two .......................................... 195
12.5 Exposing the generality to software .................................... 196
12.6 Generality in other designs ................................................ 197
Summary ............................................................................. 198
CHAPTER 12. ISSUES OF GENERALITY

12.1 Introduction

We consider two levels of generality. The first level is common to any microprocessor design: show whether the proposed design is fit for use for true general-purpose computing, and how the interactive Turing machine is approximated. This reflects the requirements set forth in sections 1.2.1 and 1.3. The second level is specific to a design that offers a parallel execution environment: define what the abstract semantics of communication and synchronization are, how new processes and channels are defined, and what are the conditions under which deadlock freedom and progress are guaranteed.

Usually, in computer architecture, the ancestry of a new component is sufficient to argue for its generality: a new pipeline that supports control flow preemption via interrupts and is connected to a sufficiently large RAM is conceptually sufficiently close to all previous microprocessor designs that its generality can be *implicitly inherited*. Usually, universal and interactive Turing-completeness for individual pipelines stems from conditional branches and the RAM interface to both storage and external I/O, whereas generality relative to parallel execution comes from the availability of time sharing and virtual channels over a consistent shared memory. However, the situation is not so clear if some features are omitted relative to previous approaches, or in the presence of hardware parallelism. For example, the proposed design from part I does away with control flow preemption and consistent shared memory: can it still be used to support general concurrency patterns? It also supports orders of magnitude more concurrent threads than in existing processor designs: what is the cost to guarantee access to a private memory on each individual thread, as required to guarantee Turing-completeness?

An argument for generality can be constructed either by showing that a new design can be described by one or more of the existing theoretical models, or can *simulate* arbitrary programs expressed using these models. Turing-completeness has the Turing machine, \( \lambda \)-calculus or queue machines, as seen in section 1.2.1. For concurrent execution, one can use Hoare’s Communicating Sequential Processes (CSP) [Hoa78], Hewitt’s Actors [HBS73] or Milner’s \( \pi \)-calculus [MPW92a, MPW92b].

In doing so, the finiteness of resources in a concrete implementation should be acknowledged. In general, when some property of an abstract model is dependent on the recursive unfolding of a theoretically unbounded inductive abstract rule, for example the arbitrary scrolling of the tape in a Turing machine, or a property over the replication operator \( \mathbf{!}P = P | P | P | \ldots \) in the \( \pi \)-calculus, the argument about an actual design must show which concrete resource supports the unfolding and which implementation parameter(s) can be tuned to approximate an arbitrarily large specification of a program in the abstract model. In particular, the simulation should be *complexity-preserving*: a basic operation of the simulated model with a constant time and space cost should have a constant maximum time and space cost in the simulation.

As parting words on the topic of the inner question around hardware microthreading, this chapter argues for this generality in the proposed innovation.

12.2 Turing-completeness: one and also many

The generality of the design in part I can be directly derived from the generality of *one* individual thread context on chip. As the proposed innovation reuses a standard RISC pipeline, including its support for conditional branches and its memory access instructions, and execution using only thread has access to the entirety of the arbitrarily large external
memory, the thread can be described by (and simulate) a universal Turing machine, with intuitive cost properties.

An issue arises when considering multiple thread contexts executing simultaneously. For each context to be Turing-complete, all must have access to their dedicated, private region of memory to simulate their Turing machine’s tape. As illustrated in chapter 9, this can be achieved by splitting the logical address space of the entire chip by the actual number of thread contexts on the chip. Other such interleavings can be constructed for any system implementation (cf. side note 12.1). If there are $N$ thread contexts, a Turing machine of any degree of complexity can be implemented by any thread context independently from all others by growing the external memory accordingly by a factor $N$. This simulation of private memories is further complexity-preserving, assuming the back-end RAM system provides independent, bounded access times to all memory locations.

Finally, we highlight that Turing-completeness is not sufficient per se; as mentioned in section 1.2.1, threads should also support interaction, i.e. the ability to interact with the outside world besides RAM. Yet the design from chapter 3 proposes that only some cores are connected to an external I/O interface. For the other cores, the argument for generality can be constructed as follows: as long as an operating software proposes an I/O forwarding service to all cores, where an I/O read or write access from any thread can be delegated, via TMU events over the chip, to a core with physical access to I/O devices, then all cores can be considered general. This simulation of interaction for all cores can further be made complexity-preserving by ensuring that the I/O forwarding services have a fixed overhead, i.e. with a maximum added I/O latency and space requirement for management data structures; this property was e.g. achieved in our implementation of POSIX system services (cf. section 6.4.2); the distributed operating systems Barrelfish [SPB+08] and fos [WA09] also support this pattern.

12.3 Abstract concurrency models

12.3.1 Prior work

- In [BBG+08, Jes08b, vTJ11, vTJLP09] and other publications, the authors introduce “SVP,” a “general concurrency model” derived from hardware microthreading. This model describes the behavior of a program in term of concurrent families of threads that are dynamically created and synchronized upon by individual threads. It also defines dataflow channels between threads directly related by family creation, and an asynchronous shared memory where the visibility of writes relative to loads is only guaranteed by family creation and synchronization.

  In [VJ07], the authors project the semantics of SVP onto thread algebra [BM07] to demonstrate deadlock freedom under at least the following conditions:

  - families are created and synchronized upon by the same parent thread;
• all actions other than reading from a channel are not blocking;
• the dataflow channels are connected so that the directed dataflow graph is acyclic, a thread may only be waiting for a data produced by its predecessors, and all channels are provisioned with a value by their source thread after the source thread has received a value from its predecessor threads.

Furthermore, [VJ07] shows that SVP is deterministic with regards to the values emitted on its dataflow channels and the values written to memory when the program is both deadlock free and race free.

Finally, SVP disallows arbitrary point-to-point communication. Dataflow channels are only connected between a parent thread and the families it directly creates. Communication via memory between concurrent threads is in turn disallowed by stating that writes by one thread are only visible to loads by itself and its successor threads: either those created as new families by the writer thread, or the parent thread after the writer’s termination. As such, SVP cannot be used to simulate arbitrary concurrency patterns from CSP, Actors or π-calculus.

This abstract model assumes unbounded resources, in particular that threads can communicate over an arbitrary number of channels and that the creation of a family is a non-blocking operation that always succeeds. As we discussed in chapters 4, 8 and 10, this assumption does not hold for the proposed hardware design, where the maximum number of dataflow synchronizers and bulk synchronizers are bound by physical resources on chip.

To increase the number of channels that can be defined from SVP’s perspective, we defined in chapter 8 a method to “escape” channels as slots on the activation records of thread programs. This simulation is also complexity-preserving since each communication operation is simulated by a fixed number of memory addresses and a fixed number of memory stores and loads. However, using this method an arbitrary number of channels can only be created if the thread context performing a family creation has access to an arbitrarily large private memory. For the scheme to work from any context, every context must have access to an arbitrarily large private memory.

Similarly, to increase the number of families that can be created from SVP’s perspective, we defined in chapter 10 a method to “escape” creation as a sequential procedure call. Again, this simulation is complexity-preserving as each family creation translates to a fixed number of memory addresses and operations. However, using this method an arbitrary number of families can only be created if the first thread context where hardware creation fails supports an arbitrary recursion depth. For the scheme to work from any thread context, an arbitrary sequential recursion depth must be available everywhere. Also, this method only works because the behavior of any family in SVP is serializable into a sequential process.

We highlight that the abstract definition of SVP in [VJ07] and other previous publications did not mention explicitly their requirement on arbitrary large private memories and recursion depths from individual SVP threads. As we showed here, these features are mandatory if SVP is to be simulated by a platform with a finite number of concurrency resources. Conversely, if an implementation does not offer arbitrarily large memory and recursion depth per thread, then it does not simulate SVP and is not free of deadlock even when all dataflow channels are provisioned. Of course, if all thread contexts are otherwise Turing-complete, for example as discussed above in section 12.2, both escape mechanisms are implementable and the system can simulate SVP completely.
12.3.2 Communicating sequential processes

We postulate that the proposed hardware design can simulate the semantics of Hoare’s CSP in the form described in [Hoa85]. Unfortunately, we have not yet demonstrated this formally; this section merely argues that the platform goes a long way towards supporting CSP, which the SVP model above does not. If the platform can run CSP programs, software audiences can gain confidence that it can also support existing programming abstractions based on CSP such as Occam [MS90], Erlang [AVWW96] or MPI [Mes09].

We consider here only CSP programs that define up to \( N \) processes, where \( N \) is the number of thread contexts available in the platform. We place this restriction because supporting more than \( N \) processes would require interleaving of multiple CSP processes over single thread contexts, which the architecture does not currently support (cf. section 3.3.2). We do not consider this as a strong restriction however: the reference platform we used for evaluation (cf. table 13.1) supports at least 4000+ simultaneously running thread contexts\(^1\). Further technology advances will likely allow the implementer to increase this limit further.

CSP further exhibits the following features:

- processes can communicate over named channels;
- communication is synchronous, i.e. sending a value in one process and receiving a value in another are a rendezvous;
- each process can name (and thus use) an arbitrarily large, but statically known set of channels;
- processes communicate values over channels, and values can be neither channel names nor processes;
- choice: processes can wait for an event from two or more channels simultaneously and react as soon as any one channel has an event available;
- channels support communication only in one direction and between two processes [Hoa85, Sect. 4.2, p. 114];
- processes can recurse and activate new processes at every step of the recursion\(^2\);  
- concurrent events that do not synchronize processes can proceed concurrently.

Per-process recursion stems from the Turing-completeness of individual thread contexts, discussed previously. The activation of new processes can in turn be implemented using the concurrency management services of the TMU introduced in section 3.3.1. Since separate thread contexts are independently scheduled (cf. section 3.2.1), concurrency of CSP processes mapped to separate thread contexts is guaranteed.

Not considering CSP’s support for choice, point-to-point communication can be implemented by the platform’s synchronizers and the remote register access primitives of the TMU, described in chapter 3. More specifically, one channel between two processes can be implemented by using two synchronizers at each endpoint, using remote writes to one

\[^1\]It actually supports up to 32000+ contexts if some of them consume less dataflow synchronizers per context than the number allowed by the ISA (cf. section 3.3.3 and chapter 8). However, the number of independent thread contexts is also bound by the number of bulk synchronizers, 32 per core in the reference implementation. Unless multiple CSP processes can be instantiated in a single bulk creation, the number of bulk synchronizers bounds the maximum number of CSP processes that can be created.

\[^2\]In his original paper [Hoa78], Hoare recognized that the semantics of CSP allowed programs to dynamically create new processes but was reluctant to encourage the use of this feature. In the later CSP book [Hoa85], dynamic process creation was explicitly allowed and a note was simply added at the end that some implementations, in particular Occam on the Transputer [MS90] could not create arbitrary numbers of new processes dynamically [Hoa85, Sect. 7.3.6, p. 225].
CHAPTER 12. ISSUES OF GENERALITY

synchronizer to transmit data and remote writes to the other for acknowledgements. This
direct mapping of CSP channels to pairs of hardware synchronizers supports at most $L/2$
channels, where $L$ is the number of private synchronizers allocated for the thread context
(cf. chapter 8); it is complexity-preserving since each CSP communication is simulated by a
fixed number of synchronizing operations.

Simple pairs of synchronizers cannot implement CSP’s choice operation. Choice cannot
be implemented over multiple synchronizers, because one thread can wait on at most one
synchronizer at a time. Choice from multiple channels cannot be implemented over a single
synchronizer either, because concurrent remote register writes to the same target register
yield undefined behavior: they are resolved non-deterministically, and the receiving thread
does not know that more than one event has been delivered, incurring event loss.

To implement choice, communication must be lifted to active messages [vECGS92] instead:
sending a message by a CSP thread over a CSP channel must be implemented on our
platform by sending a request to create a new thread at the core of the receiving thread.
The new thread runs a dedicated “remote delivery” thread program which then stores the
value, at the target core, into some incoming buffer and wakes up the thread(s) waiting
for input from the CSP channel. This simulation is intuitively complexity-preserving. This
basic idea then raises two issues.

The first issue is how to guarantee that there is always a thread context available to
receive incoming active messages. This can be obtained by pre-allocating a context on each
core prior to program start-up using the TMU’s “allocate” event, then storing the context
addresses into an array visible from all cores. Then each core willing to communicate would
look up the context address from the address of the target core, and send an active message
by sending the TMU’s “create” event followed by “sync” to the pre-allocated context. “Sync”
is required because CSP communication is synchronous: the sender thread must wait until
reception is acknowledged by the target mailbox.

The second issue is how to wake up one thread from another on the same core. To
implement this, a thread that wishes to wait on one or more channel(s) can reserve a single
synchronizer $R$ locally, then set the state of $R$ to empty, then write the address of $R$ and
the identities of the channel(s) it wishes to wait on in local memory at a location visible
from the remote delivery threads on the same core, then start reading from $R$ which causes
the thread to suspend. All these primitives can be implemented using the interfaces from
chapter 4. When a remote delivery for a given channel arrives at a core, its thread program
first reads the synchronizer addresses $R_1, R_2, \ldots$ for that channel from local memory (there
may be multiple processes waiting on the same channel), then sends TMU remote register
writes of the incoming value to all $R_1, R_2, \ldots$ addresses via the core’s loopback NoC interface.
Atomicity of access to the memory structures between threads on one core can be negotiated
using any of the mechanisms introduced in section 14.1.

The mechanisms introduced above effectively implement arbitrary virtual channels be-
tween thread contexts, using one mailbox per core that can be implemented in a local
memory. It therefore lifts the limitation on the number of channels introduced above by the
direct use of synchronizer pairs. The total number of channels that can be read from in a
process is only bound by the size of the local memory at the process’ core.

To summarize, the proposed platform is likely as powerful as Hoare’s CSP, and our
proposed simulation of CSP uses only the concurrency control primitives offered by the
TMU without requiring a single memory system shared by all cores. It is also universal
since the definition of processes and channels can be described fully in software.
12.3.3 Actors

In contrast to Hoare’s CSP, the Actor model introduced by Hewitt [HBS73] and developed by Agha [Agh85] is relatively simple:

*Actors are computational agents which map each incoming communication to a 3-tuple consisting of:*

1. a finite set of communications sent to other actors;
2. a new behavior (which will govern the response to the next communication processed); and,
3. a finite set of new actors created.

When an actor receives a message, the system spawns a task which computes this 3-tuple. The Actor model does not mandate a specific computational power for the implementation of individual tasks, although [Agh85] suggests that tasks can at least perform arithmetic and simple conditionals.

Communication between actors is further defined to be asynchronous, that is, an actor is immediately ready to accept a new message after sending a message irrespective of whether the target actor(s) have already processed the communicated event. An actor can only communicate with another target actor if it previously knows its mail address, i.e. there is no global name space of actors. Each new actor causes the appearance of its own new mailing address, which is known to the creating or created actor, or both.

Following the discussion on CSP above, we can argue that the proposed platform from part I can implement a system of actors. To describe a single actor, an implementation can use a data structure on a local memory on one core which contains a pointer to the task program and its initial parameters, provided when the actor was created. An actor mail address is then composed by pairing the address of the core and the local address of the actor data structure on that core. Then the mailbox mechanism introduced in section 12.3.2 is reused: when a task wishes to send a message to a actor whose address it knows, it sends a “create” TMU event to the target core, providing the local address of the actor’s structure as functional argument of the “create” message, and the pointer to a “remote delivery” thread program. The task needs not use “sync” since actor communication is asynchronous. On the target core, the “remote delivery” thread program in turn looks up the actor structure from the provided address in its local memory, then creates (possibly locally) a new thread running the task program specified in the actor structure. It also provides as arguments to the task both the initial actor parameters and the actual payload of the received message. The remote delivery thread can then terminate without waiting on the created task.

To define the “next behavior” of its actor, a task simply overwrites the task and parameters in its own actor’s data structure.

Finally, a task that wishes to create a new actor simply requests allocation of an actor data structure from the environment. In [Agh85] the author leaves open where the environment should create the actor physically, so an implementation on our platform could choose any core on the system that has some free local memory remaining.

Using these mechanisms, the maximum number of actors that can be active at the same time is bounded by the number of independent thread contexts, e.g. 4000+ in our reference implementation. However, the maximum number of actors that can be defined, as well as the maximum number of messages in-flight between actors, is bounded only by the total capacity of the private memory reachable by each core in the system. This can either be
bounded by on-chip memory capacity if cores use local scratchpads, or become arbitrarily large using the techniques from chapter 9. This simulation is complexity-preserving for the behavior of individual actors if the actor tasks are run natively by hardware threads. It can also be made complexity-preserving for communication if a bound is set on the maximum number of actors, so that the look up process at each mailbox has a bounded maximum space and time cost.

To summarize, we can construct a simulation of the Actor model on the proposed platform using only TMU primitives and local memories at each core. Our simulation fully exploits the opportunities offered by actors for concurrent execution, both from the behavior independence of distinct actors and the asynchrony of communication. It is furthermore universal since actors can be described fully in software.

### 12.3.4 π-calculus

Milner’s π-calculus is a development from Milner’s own Calculus of Communicating Systems (CCS) [Mil80], developed concurrently to Hoare’s CSP.

Because CCS and CSP where both inspired from the zeitgeist of their times, they share a number of similarities: processes communicate over channels, communication is synchronous, processes can be combined concurrently or via the choice operator, and recursions of concurrent process definitions are allowed. One of the main differences between CCS and CSP is that CCS offers a restriction operator, which allows a group of processes to bind a channel name to a private channel not visible by other process groups. In contrast, in CSP the channel name space is shared by all processes.

As in CSP, in CCS channels and processes are in name spaces distinct from the values communicated over channels. The main extension of the π-calculus over CCS was to allow channel names as communicated values. Moreover, the π-calculus supports arbitrary N-to-M communication over single channels.

As in section 12.3.2, we postulate that the proposed platform from part I can simulate the semantics of the π-calculus as described in [MPW92a, MPW92b]. Again, we have not yet demonstrated this formally but we are able to argue so based on simulations.

We start by observing that we cannot directly reuse the simulation infrastructure introduced in section 12.3.2. Although our proposal for CSP can be trivially extended to support a dynamically evolving number of channels, and although the identity of channels could be communicated as values, it binds the receiver endpoint of a channel to a specific core address on chip, where the receiving process is running. This is needed because the active message implementing remote delivery must be addressed to an explicit location on chip. Moreover, so far we considered only one sender and one receiver per channel. In contrast, in the π-calculus, any process can send or receive from a channel whose name it knows. Our previous mailbox-based system which assumes that receiving threads are on the same core as the mailbox does not support this.

Instead, we can implement a forwarding service as follows. Any time a process in the π-calculus defines a new channel, its implementation as a thread would send a request at a commonly agreed “channel management” service in the system that would allocate a new channel data structure in the local memory of an arbitrarily selected core in the system to serve as shared mailbox. Any subsequent operation that sends a channel identity would then communicate the address of the shared mailbox to the receiver process(es); any time a process receives a channel identity, it would receive the address of the corresponding shared
mailbox service and would then inform the shared mailbox that it is now a candidate receiver for that channel. This technique is inspired from the IPv6 “home routing” protocol [PJ96].

Subsequently, any communication over the channel would cause two communications, one from the sender thread to the shared mailbox, then from the shared mailbox to the local mailbox of the candidate receiver process(es).

This mechanism indirects communication between two threads implementing $\pi$-calculus processes via a third party core, and thus incurs an extra latency compared to the simulations of sections 12.3.2 and 12.3.3. However, each new channel can be instantiated over a different shared mailbox, and the channel management service can thus theoretically spread the communication load over the entire system. This simulation is intuitively complexity-preserving for the sequence of actions by individual processes. As with actors above, it can be made complexity-preserving for communication if a bound is set on the maximum number of channels, so that the look up process at each mailbox has a bounded maximum space and time cost.

To summarize, the proposed platform is probably as powerful as Milner’s $\pi$-calculus. As with CSP above, our proposed simulation uses only the concurrency control primitives offered by the TMU without requiring a shared memory system. It is also universal.

12.4 Turing completeness, take two

In [Mil90], Milner considers pure $\pi$-calculus, where the only actions that can be performed by processes are the communication events and operators of the $\pi$-calculus itself. The author then proceeds to demonstrate that the pure $\pi$-calculus is equivalent to Church’s $\lambda$-calculus, by simulating two evaluation strategies for $\lambda$-terms in the $\pi$-calculus (one strict and one lazy). His simulations implement $\lambda$-terms as processes in the $\pi$-calculus, and environment bindings of variables to $\lambda$-terms as replicating processes that forward the binding information to other requesting agents implementing $\beta$-reductions. Partial $\beta$-reductions further execute as concurrent processes in the $\pi$-calculus, and thus fully take advantage of the inherent concurrency of the $\lambda$-calculus.

With this proof in hand, we can consider a hypothetical implementation of the proposed architecture where some individual thread contexts are not Turing-complete. For example we could reduce the implementation so that some thread contexts have access only to a small amount of local memory, invalidating the techniques from chapter 9. Or we could consider that some thread contexts can use only a smaller instruction set which does not support arbitrary recursion depths. In this heterogeneous platform, we would then be able to distinguish between “fat,” general-purpose thread contexts and “light” thread contexts with reduced functionality.

In this setting, the simulation techniques presented in section 12.3.4 can still be used to simulate the $\pi$-calculus using “light” contexts. Indeed, the primitives from the section 12.3.4 are implemented in our simulation using only simple synchronizing reads and writes and sending TMU events. The main simulation logic occurs outside of the simulated $\pi$-calculus processes, in the mailboxes, where larger memory and recursion are required. However, we can note that only one mailbox per core is necessary; an arbitrary number of “light” thread contexts can be used around each mailbox without changing the semantics of the simulation.

In such an environment with a few “fat” mailbox thread contexts and many “light” thread contexts, the system would be able to carry out the reduction of arbitrary $\lambda$-terms as per [Mil90], by spreading the evaluation over a dynamically defined, arbitrarily large network of small communicating processes. In other words, the system as a whole would be
Turing-complete. Moreover, its expressivity power could expand arbitrarily by increasing the number of “light” thread contexts, at a fixed “fat” context budget. The system would also be universal because processes and channels would still be specified by software. The only drawback of reducing the number of “fat” contexts is that they are a bottleneck for communication: a smaller number of “fat” contexts implies less available bandwidth overall.

Unfortunately, we were not able to determine yet whether this simulation would be complexity-preserving, that is whether a bound can be set on the space and time costs to simulate one execution step of the simulated Turing machine.

12.5 Exposing the generality to software

The arguments above merely suggest that the design is general. There are two ways forward to convince audiences more thoroughly. The first is to describe the semantics of the hardware interface from chapter 4 precisely in an abstract model, and then prove formally that this model is at least as expressive as other existing models. This approach, while necessary to gain credibility in theoretical circles, would be however largely inefficient in making the invention accessible. Instead, more popular audiences will require programming languages and frameworks that expose the system’s potential to their creativity (cf. section 1.3).

So far, our contributions provide a general-purpose programming environment for individual cores (the C language), and a handful of concurrency-related features:

- constructs to spread work over multiple thread contexts and cores, with semantics related to the SVP model described in section 12.3.1;
- a memory consistency model where updates by unrelated concurrent threads may be visible to each other (namely, if they are part of the same CD, cf. chapter 7);
- the foundations of a finite resource model (chapters 10 and 11).

These features makes our contribution only barely more powerful than the SVP model described in section 12.3.1. In particular, to implement the CSP, Actors and π-calculus simulations introduced in section 12.3, the language interface must be further extended with operators to:

- take the address of a synchronizer;
- issue a remote access to a synchronizer whose address is known;
- reserve a thread context (“allocate”) and store its identity;
- create threads in a previously reserved context;
- synchronize on termination of threads without releasing the context;
- organize atomicity of access to memory between threads running on a single core.

We estimate that little effort is required to add these primitives to the framework introduced in chapter 6, since these primitives are already available in the hardware interface from chapter 4. However, we also highlight that any program that would subsequently use these features would not necessarily be serializable any more, limiting the user’s ability to troubleshoot programming errors (cf. section 6.2.4). This may warrant the separation of these additional services in either a separate system language, or a set of privileged language constructs that would be only usable by the operating software of higher-level parallel languages, and not directly by application programmers. Alternatively, an emulation of these

3Currently, reservation and creation are bound in a single creation construct.
4Currently, synchronization and release are bound in a single synchronization construct.
primitives could be implemented on a legacy platform to serve as reference for troubleshooting. We suggest that this exploration in language design be performed in future work.

12.6 Generality in other designs

Most contemporary SMP chip designs inherit their generality in an almost boringly simple way. For example, each individual core in Intel’s general-purpose multi-core offerings, including the P6, NetBurst, Core, Nehalem, Atom, Sandy Bridge micro-architectures, combine traditional general-purpose ISAs with a timer interrupt, coherent views on a common shared memory and direct access to system I/O; so do AMD’s K8-K10 and Sun’s/Oracle’s SPARC multi-core products. The timer interrupt in turn allows programmers to define arbitrarily large numbers of virtual processes interleaved using a software scheduler; the coherent shared memory enables arbitrarily large numbers of virtual channels connected in arbitrary patterns. Universality and interactivity on each core are evident, as every behavior is defined in software and all cores have symmetric access to I/O. We can find more diversity in other “exotic” designs that have surfaced in the last 10 years.

In IBM’s Cell Broadband Engine [KDH+05], a general-purpose PowerPC core called PPE is combined with 8 to 16 smaller RISC cores called SPEs. Although each SPE supports a general instruction set featuring conditional branches, and direct access to the outside world via its own Direct Memory Access (DMA) controller, it is connected only to a local RAM with a capacity of 256KiB which contains both code and data. This capacity may be sufficient to accommodate the workload of sub-computations driven from the PPE; however it seems to us insufficient for general-purpose workloads driven directly from the SPE. Meanwhile, each SPE supports inter-SPE communication via 128 distinct synchronous channels implemented in hardware. This makes the SPE group a suitable platform for simple process networks with up to 8-16 processes (the number of SPEs). Although each SPE supports control flow preemption via external interrupts, this feature could not simply be used by a software scheduler to virtualize more processes because the channel read and write operations are not interruptible.

In NVIDIA’s Fermi [LNOM08, NVI09] GPGPU designs, each core, called an SM, is equipped with a threading engine able to schedule multiple independent threads concurrently. Each thread can run arbitrarily patterns of conditional branches and can be configured to access an arbitrarily large private memory via a unified cache to external RAM (a feature not present in NVIDIA’s previous GPGPU offerings). As such, each thread features Turing-completeness. However, Fermi threads lack the generality required in section 1.2.1: they cannot be interactive, because Fermi does not allow GPGPU threads to access external I/O devices.

In terms of concurrency patterns, Fermi threads do not support control flow preemption via a timer interrupt and thus cannot multiplex multiple logical processes over one thread context. However, Fermi does support inter-thread synchronization within one SM using atomic accesses to a local, 16KiB shared memory. Since each SM can run 768 separate threads, the chip should thus support CSP and $\pi$-calculus up to that number of processes within each SM, using mechanisms similar to those we propose later in sections 12.3.2 and 12.3.4, with the number of channels limited by the local memory capacity. When considering the entire chip of multiple SMs instead, communication between SMs is possible via the external RAM, however memory atomics do not cross SM boundaries so synchronization would require busy waiting. To summarize, Fermi’s architecture can support general
concurrency patterns efficiently within one SM, and less efficiently across all SMs in one chip.

We also considered Intel’s Single-Chip Cloud Computer (SCC) [MRL+10], a research platform, and Tilera’s TILE64 [BEA+08], a product offering for network applications. Both integrate a larger number of general-purpose cores on one chip than contemporary multi-core product offerings: 48 for the SCC, 64 for TILE64. All cores are connected to a common NoC. On both chips, Turing-completeness at each core is achieved by a traditional design—the MIPS pipeline for TILE64, the P54C pipeline for the SCC—and a configurable mapping from cores to external memory able to provide the illusion of arbitrarily large private memory to each core. Interactivity is provided on the TILE64 by direct access to external I/O devices on each core via dedicated I/O links on the NoC; on the SCC, the NoC is connected to an external service processor implemented on FPGA which forwards I/O requests from the SCC cores to a host system. The SCC approach to I/O is thus similar to the one we took in chapter 5.

For parallel execution, TILE64 and SCC only support one hardware thread per core, but cores feature preemption as the means to multiplex multiple software processes. TILE64 offers comprehensive support for communication: it supports 4 hardware-supported asynchronous channels to any other cores (UDN), a single channel to a configurable, static set of peers (STN) and two dedicated channels to external I/O devices per core (IDN). Alternatively, cores can also implement virtual channels over a coherent, virtually shared memory implemented over another set of NoC links (MDN), although the communication latency is then higher. This diversity of communication means ensures that the design can support most general parallel programming patterns.

In contrast, the SCC does not offer a coherent view of shared memory to cores. While each pair of cores has access to a local scratchpad of 256KiB, called Message-Passing Buffer (MPB), which can be accessed remotely by other cores via the NoC, the MPB does not synchronize. Instead, point-to-point synchronization can be negotiated only via IPIs, or by disabling the local caches and busy waiting on changes to external memory regions. As such, while the SCC theoretically supports most general parallel programming patterns, its actual implementation yields poor point-to-point communication latencies.
Summary

When designing new components as building blocks for computing systems, the innovator should describe the level of semantic generality provided by the invention. Especially when designing components for general-purpose systems, generality should be argued by relating the new component to theoretical models whose generality has been previously established. In most contemporary microprocessor designs, generality is implicitly inherited by reusing the traditional model of Turing-complete processors implementing timer-driven control flow preemption and connected to a shared memory that can implement arbitrary point-to-point communication. These basic conceptual models inherit Turing completeness from the individual processors and the semantics of most abstract concurrency models developed since the 1970’s.

In contrast, the proposed CMP design that we covered earlier does not provide preemption. It supports but does not require a shared memory. Also, it provides a large number of thread contexts which compete for a single address space. Because of these differences, a new bridge must be constructed between this design and existing abstract models before it can be advertised as “general.”

- In this chapter, we have acknowledged previous work in this direction by our peers, where a “concurrency model” named SVP was defined. We also showed the limitations of this approach.

- Instead, by constructing simulations using only the platform’s dedicated hardware concurrency management primitives and private memory on each core (i.e. without requiring a shared memory system), we were able to relate it to Hoare’s CSP, Hewitt and Agha’s Actors and Milner’s π-calculus. Furthermore, using our contribution from chapter 9, we were able to regain Turing-completeness for individual cores under the assumption of arbitrary large external memory. If this latter assumption does not hold, we are still able to suggest Turing-completeness for the entire system based on support for the π-calculus. Our simulations are based on the machine primitives described in chapter 4. These primitives are not all yet exposed in the language interface from chapter 6; therefore, future work must provide further language support before the full generality of the platform becomes available to external audiences.