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 14

System-level issues
—Lessons learned on and about the way to integration

Abstract

During interactions with an audience, the platform provider may gain additional insight about the innovation from the interaction process itself, independently from the audience. In this chapter, we provide an example of this. During our implementation work for the platform in part II, we discovered additional system-level issues not directly visible to our audience when carrying out the evaluation work reported on in chapter 13. Again, as predicted in section 5.3, these issues translate directly to feedback about the architecture design. We document them here.

Contents

14.1 Mutual exclusion .................................................. 224
14.2 Event and fault management ................................... 225
14.3 Virtual addressing trade-offs ................................... 227
14.4 Implicitly carried state ........................................... 228
14.5 Process state, termination and reclamation ................. 232
14.6 Feedback from education activities ............................ 233
Summary ................................................................. 233
14.1 Mutual exclusion

To manage shared resources, the platform must provide a means for the operating software to negotiate *atomic transactions*, or mutual exclusion, over these resources from the perspective of concurrent activities.

In the case of physical I/O devices that have specific locations on the chip, mutual exclusion can be achieved by forcing the serialization of work at the locus where the physical interface exists. For example, in our work, the processors physically close to the I/O devices feature a unique thread context whereby remote requests to create work at that context would not be accepted until the previous work has terminated. This is equivalent to the “secretary” concept from [Dij71, p. 135], and transparently reuses the existing mechanisms from chapters 3 and 4.

To control access to *logical* shared resources like heap allocators or logical queues, existing software requires the availability of *logical synchronization devices* like monitors, locks or semaphores. How these are implemented, whether via memory transactions over a cache coherency protocol, or dedicated synchronization messages on a NoC, or a combination of both, seems to us orthogonal to the architecture design principles from chapters 3 and 4. Nevertheless, we were asked to make an educated commentary about the opportunity to implement all logical synchronization devices using only the hardware-based serialization mechanism described above. The motivation for this is to alleviate the need for global coherency on the memory network (cf. section 3.4.1).

We can comment on this opportunity with two observations.

The first observation is that an efficient implementation of a back-propagation mechanism of a contention situation to the locus of issue requires attention throughout the chip. To understand why, consider first that proper *suspension* of a thread, in a manner that lets other threads progress, requires to store the identity of the thread *somewhere*, i.e. in a dedicated memory structure with a physical location. This state must then wait *passively* for a wake-up event. This implies that a thread can only be woken up upon reception of an explicit message by the locus where its continuation is stored. In the mutual exclusion scenario, if a thread attempts to access a shared resource already in use, it must suspend; conversely, when the shared resource is *released*, the resource must notify all physical locations that contain suspended threads waiting on the shared resource. For example, with the physical serialization outlined above, threads must suspend upon writing a creation request over virtual channels to a contended shared resource; when the shared resource becomes available, it must wake up, via flow control over all virtual channels to the resource, the remote threads that were previously blocked.

A potential naive implementation of this necessary notification is based on the traditional concept of “waiting thread queue,” where each shared resource would store locally a dynamically evolving set of client loci containing thread continuations to be resumed upon release. This would in turn require a local memory whose size is linear with the total number of loci in the system which may participate in the mutual exclusion. If any core must be able to host a shared resource, this implementation would require a space requirement quadratic in the size of the system. Instead, to minimize the overhead, a tree-like notification network of virtual channels can be built from the shared resource to all candidate clients, using flow control to control access to the shared resource. This in turn requires implementation of flow-control fan-in at all branching points on the NoC with a prioritization scheme to avoid starvation, multiplied as necessary if any core must be able to host a shared resource.
14.2 EVENT AND FAULT MANAGEMENT

The trade-off is therefore a choice between increased storage costs at each locus, or a coordinated protocol between all loci which must in turn be made resistant to network contention and faults, even when these originate from loci not participating in the mutual exclusion.

The other observation is that it forces every access that queries the state of the shared resource, even when it is not contended, to place a roundtrip across the NoC on the critical path through execution. The latency of this roundtrip may become arbitrarily large depending on other activities on the NoC, even if the target shared resource is not contended. This violates a common expectation about logical synchronization devices, that they should have a local latency when they are not contended\(^1\) [MCS91]. To avoid this situation, dedicated Quality of Service (QoS) protocols must be used on the NoC, in turn increasing implementation costs.

To summarize, support for logical shared resources requires support from the architecture to negotiate mutual exclusion. Whether new dedicated hardware solutions are designed, or existing synchronization mechanisms over cache coherency protocols are reused, any solution must be scalable: not incur interference between unrelated activities and avoid mandatory overheads for non-contended resources. Beyond our suggestions from section 13.8, the corresponding mechanisms are not yet clearly defined in the architecture design from chapters 3 and 4 and must thus be addressed in future work.

14.2 Event and fault management

As soon as work starts to define a full system, the need arises throughout to handle unexpected conditions. Here we exclude software exceptions that can be implemented cooperatively using software techniques only, and those hardware faults that can be hidden from software entirely (via redundancy, hardware error correction, etc.). We further consider separately traps, which are caused by programs as an effect of performing particular actions (e.g. divide by zero, address translation fault) and asynchronous events which are not expected to occur at specific points during execution, but are still expected to occur sometimes (e.g. availability of out-of-band input on a communication channel, termination of a child process). The latter must be supported in particular before a system can be advertised as an interactive general-purpose platform (cf. section 1.2.1).

Traditionally, the common mechanism used to report both traps and asynchronous events is to interrupt an instruction stream, and transfer control of its execution preemptively to a configurable event handler which may then decide to either address the exception and transfer control back to the program, or trigger an alternate behavior, including possibly program termination. In contrast, in a parallel setting with potentially many threads of execution, a diversity of solutions may be considered.

One alternative, used in the POSIX threading API, is to report traps via interrupts to the specific thread that causes them, and report asynchronous events via interrupts to one, non-deterministically chosen thread among those currently executing threads that have declared their readiness to be preempted via sigaction and sigsetmask. There are two known shortcomings to this approach that limit its scalability to larger parallel systems. One is that every event in the system, including non-local communication events, must encode the identity of its causal thread somehow so that an error can be reported accurately.

\(^1\)e.g. the time to acquire a lock that is not previously locked, or to decrease a semaphore with a counter still greater than 1, should stay minimal and independent of the overall system activity.
While this is cheap to achieve for FPU exceptions, it may cause a burden on the hardware with e.g. accesses to virtual memory when address translation is resolved by a common MMU shared among many cores, or accesses to I/O channels when events may be detected topologically far from the locus of thread execution. The other shortcoming is that this form of reporting requires every thread to have access to an area of storage where its current state can be saved at the point the signal handler is invoked. As we have explained in chapter 9 this is a non-trivial requirement as the number of threads grows.

There several other alternatives possible.

The designers of UTLEON3 propose in [DKK+12] to stop the execution of all currently running threads, and trigger the execution of a high-priority “handler thread” that would run to completion before execution of the other threads resumes. This mechanism was also suggested earlier by the author of [van06]. This mechanism is intuitively appropriate to handle asynchronous events, or to cause termination upon traps, but it does not let software react to salvageable traps such as floating-point exceptions. In [van06], we can find an extension of this solution which proposes to:

- report traps to their causal threads by suspending the causal thread, triggering execution of a separate, fault handling thread on the same core, and have the fault-handling thread automatically receive from the hardware, as input parameter, the identity of the causal thread. Upon resolving the behavior, the fault-handling thread could then choose between resuming execution of the causal thread or trigger some alternate behavior like program termination.
- for asynchronous events, simply create new, regular threads for each event received.

Both with this solution and the UTLEON3 solution above, the reason why interruption of the control flow cannot be used is that the visible synchronizer window layout (cf. sections 3.3.3 and 4.2 and chapter 8) expected by the fault handler may not be compatible with the window layout of any of the currently executing threads. Note however that this physical separation between causal threads and fault handling threads does not imply that the fault handler cannot impact the control flow of the causal thread: the hardware interface could provide a primitive to the fault handler to override the PC of the causal thread and write its registers remotely. This would in turn be sufficient to allow individual threads to register different exception behaviors in software, and implement e.g. C’s `longjmp`, which is the traditional mechanism to override control flow from asynchronous handlers.

A more interesting issue related to the creation of new threads is that existing software frameworks often assume a consistent view of all the program’s variables when an event handler is invoked. The corresponding issues from chapter 7 notwithstanding, with a large many-core system this expectation would require global synchronization of the memory state at every event delivered, in turn significantly impacting performance. While global synchronization could be envisioned for rarely expected faults, it seems too expensive to require for general asynchronous events such as notifications for external input.

Researchers in operating system and language design may see here an opportunity to redefine the software abstractions for handling unexpected events, including abandoning the vocabulary related to “interrupts” and “control flow preemption.” A future-oriented solution may include abandoning precise exceptions and global view on the program state altogether, and acknowledging that the physical location of the execution of an event handler on chip will impact how much of the program’s state it can observe. However, embracing this conceptual revolution so early would be ill-advised, since, as we explained in chapter 5,
14.3 VIRTUAL ADDRESSING TRADE-OFFS

Figure 14.1: Possible locations for the address translation logic on chip.

integration in existing ecosystems requires solutions that preserve current abstractions, at least in an initial phase.

Instead, we propose as an alternative approach to exploit the companion processors that we introduced in section 5.5.1. For this, we first differentiate between programs that explicitly manage exceptions, recognized by their explicit use of the signal and sigaction APIs, from those that don’t. The programs that use the API are then constrained to always execute on the companion processors, where their threads can be interrupted in the traditional way. The programs that don’t use the API are not constrained and can be mapped anywhere on the chip; any system exceptions they receive are then handled in the way compatible with the default POSIX behavior, i.e. either by ignoring them (e.g. SIGIO, SIGURG) or by causing termination of the entire program (e.g. SIGINT, SIGSEGV). This solution preserves past assumptions for programs that exploit them, while letting assumption-free programs take advantage of all the architecture’s features. However, we did not evaluate this solution experimentally and cannot report on its implementation costs.

Regardless of which solution is eventually adopted, we highlight that the mechanisms for handling unexpected events must be duly documented and illustrated before the proposed architecture can be used as a general-purpose substitute by its envisioned audience.

14.3 Virtual addressing trade-offs

In the proposed target ecosystem from section 5.4.1 and others, support for virtual address translation is expected from the hardware to implement storage virtualization and process isolation.

We were asked to make an educated commentary about the spectrum of choices about where on chip to perform translation, which we illustrate in fig. 14.1: either translation can occur at the locus where memory requests are issued, i.e. close to the cores, or it can be centralized and shared between multiple cores. Centralization is possible because, with the generalization of 64-bit wide addresses, the address space may be sufficiently large to host
the data manipulated by all concurrent activities in the system. Meanwhile, isolation can be
provided using capabilities [JLI98] over regions of the shared address space. This is the view
taken by Opal [CBHLL92], Mungi [HEV*98], Mondrian [WCA02] and other Single Address
Space Operating Systems (SASOSs). For example, in our work, the implementations for
the design from chapters 3 and 4 perform address translation at the chip boundary with a
distributed MMU, one per DDR channel. We comment with two observations.

The first is that most target ecosystems, in particular those identified in section 5.4.1,
have historically established a strong conceptual binding between process boundaries and
virtual address spaces, for example the expectation that each process has its own “private”
address space to storage. To preserve this mindset, a system implementation that shares
translation management between multiple cores, such as the platform considered, must
either:

- ensure that processes with distinct address spaces are segregated to separate locations
  on the chip, so that the physical source of a memory request is sufficient to distinguish
  its originating process, or
- explicitly communicate with every memory request the identity of the issuing process.

The former approach would be detrimental to utilization and performance when there are
more processes active than translation domains, even if the total number of processes is
smaller than the number of cores. The latter approach is costly in either hardware logic or
performance as larger mapping caches (e.g. TLBs) will be needed to support larger numbers
of processes. These trade-offs are analogous to the opposition between virtual vs. physical
\textit{cache} indexing discussed in [CD97a, CD97b].

The other observation is that any choice as to where address translation occurs must be
combined with a choice as to where process boundaries for isolation are checked. Process
boundaries are typically checked by system services, such as resource managers, to ensure
that separate processes obtain separate views on the service. In presence of virtual address
translation, each entity that checks process boundaries must be able to look up data in
memory \textit{on behalf} of a requesting process, i.e. relative to the private address space of the
requesting process. This in turn implies that system services are also “clients” to translation
and participate in the design trade-offs identified above.

To summarize, we cannot determine whether centralization on the chip of address trans-
lation and isolation checks can yield significant benefits for the target ecosystems without
looking quantitatively at the number of separate processes that would be defined. Neverthe-
less, we are able to argue that address translation and the definition of process boundaries
must be handled in the same design discussion, because the system services that check
process boundaries are also clients to address translation.

14.4 Implicitly carried state

The standard C library, like most traditionally sequential APIs offered in existing general-
purpose systems, depends on \textit{implicitly carried state} invisible from programs. We detail this
observation below in sections 14.4.1 and 14.4.2, as well as its likely impact on system design
in section 14.4.3.
### 14.4. IMPLICITLY CARRIED STATE

<table>
<thead>
<tr>
<th>State</th>
<th>Carried by</th>
<th>Reset by</th>
<th>Averted by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of opened file descriptors</td>
<td>open, close, exit</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>(system)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>File positions</td>
<td>read/write</td>
<td>seek, lseek</td>
<td>pread, fwrite, aio_*</td>
</tr>
<tr>
<td>Set of opened streams</td>
<td>fopen, close, exit</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>(C library)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status of standard streams (stdin,</td>
<td>puts, printf,</td>
<td>N/A</td>
<td>Explicit stream operands</td>
</tr>
<tr>
<td>stdout, stderr)</td>
<td>getchar, scanf, etc.</td>
<td></td>
<td>(fputs, fprintf, etc)</td>
</tr>
<tr>
<td>Error status (errno)</td>
<td>All APIs related to</td>
<td>(program)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>system interfaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtoa allocator's internal state</td>
<td>dtoa, (f)printf</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>malloc's internal state</td>
<td>malloc, free,</td>
<td>N/A</td>
<td>Anonymous mmap</td>
</tr>
<tr>
<td></td>
<td>realloc, calloc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set of memory regions visible by</td>
<td>mmap, sbrk, shmat</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Environment variables (environ)</td>
<td>getenv, setenv</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Registered destructors</td>
<td>atexit, exit (C</td>
<td>N/A</td>
<td>_Exit</td>
</tr>
<tr>
<td></td>
<td>library)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point environment</td>
<td>Math functions</td>
<td>fetsetenv</td>
<td>#pragma STDC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FENV_ACCESS OFF</td>
</tr>
<tr>
<td>strtok buffer</td>
<td>strtok</td>
<td>strtok</td>
<td>strtok_r, strsep</td>
</tr>
</tbody>
</table>

Table 14.1: Implicitly carried dependencies in C/POSIX APIs.

### 14.4.1 Problem statement

The assumption of sequential program execution has traditionally enabled API designs with implicitly carried state from one API call to another. For example, POSIX’s `read` and `write` calls carry the current I/O position in the file implicitly from one call to the next. Due to their history as an environment for mostly sequential programs, both the C language, the POSIX interfaces and other API framework for general-purpose programming languages are rife with sequential APIs; we list the most common from Unix in table 14.1.

There is no theoretical difficulty here: a given sequential programming interface is either known to be appropriate for concurrent calls (this is documented as “reentrant” or “threadsafe”), or it is not. Practically, this difference determines who is responsible for ensuring proper synchronization: in the former case, the service provider must provide synchronization, and in the latter case the application programmer must ensure that calls are synchronized. Race conditions occur when neither take this responsibility, and over-synchronization occurs when synchronization is negotiated on both sides of the service interface.

What we found interesting however, is the impact of this semantic mandatory sequentialization on multithreaded performance: regardless of the overall structure of the program, how it organizes concurrency between components, and how API services are implemented,
the set of all effective API calls to sequential APIs during execution constitutes an *implicit sequential path*, invisible from the explicit program description.

We illustrate this in fig. 14.2: even when threads of distinct application components are independent from each other, an implicit synchronization occurs every time they issue a call to a common API. The API service, as a participating entity during the execution, is in fact a *shared resource* which can cause contention.

In general terms, the implicit sequential path of a sequential API becomes part of the *critical path* of execution, and thus a constraint on performance, if the overall *issue rate* of API calls across all threads to the API provider becomes equal to the *maximum bandwidth* of the API provider.

14.4.2 Contended resources

In our setting, the large amount of concurrency available in hardware allows us to increase the issue rate to a sequential API far beyond the bandwidth of single cores where the API runs. We observed concrete occurrences of contention caused by sequential APIs in the following circumstances:

- dynamic memory allocation: calls to the `malloc` and `free` APIs are frequent, for example in code from higher-level programming language like SAC [GS06]. They cause contention if served by a single heap allocator. We were able to reduce this effect partially by using a concurrent heap allocator, however larger numbers of concurrent heaps in turn cause contention on system-level dynamic allocation of physical RAM pages to back the virtual address space;
- memory reclamation: to enable garbage collection of unreferenced arrays in SAC programs, we developed jointly with our project partners a deallocation strategy based on reference counting. This strategy uses a system service for mutual exclusion to protect concurrent accesses to the reference counters. In this work, we found that the set of updates issued by independent threads to single reference counters cause contention on highly parallel workloads. We document this in [HJS+11];
- dynamic processor allocation: to enable dynamic placement of application components to processors, we implemented an allocation API for processors that programs can use
to reserve and release entire clusters of processors at once, using the abstractions introduced in [JPvT08] and chapter 11. Since this allocator must keep a map of which cores are allocated as implicit state, we also found that it becomes a source of contention on dynamically heterogeneous workloads.

In contemporary applications running on other platforms, this source of contention is usually averted in either of three ways:

- either the issue rate (average number of calls per unit of time of all threads) is kept low, by either having few threads running overall or concentrating most API calls in a “control thread”; or
- process boundaries are introduced between application components, so that the sequential APIs in each process can be served by separate service providers in operating systems (for example, operating system kernel components running on separate processors); or
- programs avoid using sequential APIs altogether, substituting uses of distributed services instead.

We highlight that explicit control of API issue rates is impractical in most programming languages and that process boundaries increase the communication latency between application components. Therefore, any new system implementation which aims at providing a general-purpose platform must provide contention-less equivalents for at least these services listed in table 14.1, for these are the services in pervasive use across most current applications and language run-time systems. Here we would like to suggest further that major abstractions in existing operating systems will be altered during this process.

To support this statement, we take the example of file I/O. The most intuitive carried state of read/write is the file cursor at which I/O operations are effected. It is possible to remove this carried state as in pread and pwrite, which take the file position as an explicit argument. Unfortunately, the file descriptor remains: every POSIX call which manipulates file descriptors must check whether the descriptor is valid and which object it designates in the virtual file system; this mapping of descriptors to objects is a carried state updated by the global sequence of calls to open and close. To remove this state, it is necessary to design the file access API to use stateless file descriptors, which do not need to be explicitly opened/closed. Although this feature is yet rarely used directly in applications, it is already available in popular distributed file systems (e.g. Sun’s Network File System (NFS) [SGK+85], Hadoop [SKRC10]) and we predict its increased use in disfavor of Unix’ traditional numeric file descriptors. This may have a dramatic impact on other APIs, such as select, or otherwise program algorithms which assume descriptors numbered contiguously from 0.

14.4.3 General problem and consequences on hardware design

While single-machine operating systems with a distributed internal structure are a mature and well-studied field of research [TVR85, TR86, SJ96, TS06], we found that the focus of most previous research on internal system data structures was on state isolation for security, both to increase accountability and fault tolerance. Comparatively, little research has been carried out to optimize performance via distribution within operating systems until recently [SPB+08, WA09].
Instead, we found previous work on performance scalability at the level of networked application services. An exhaustive and comparative review of application-level schemes are available in [Fie00], whose author revolutionized the exploitation of applications over the Internet by popularizing REST protocols: client-server, stateless, cacheable, and layered. Modern distributed file systems provide REST protocols; we suggest that future machine level services in operating systems should feature REST protocols as well to implement logical, software-only services.

The limit of the REST vision, however, is reached with shared physical resources such as address spaces in storage or external I/O ports, because the physical reality of their implicitly carried state cannot be moved around the system via logical messages.

We can predict a mandatory clustering of physical resources as a consequence. As the amount of general-purpose concurrency in application increases, so does the concurrent use of system services; a given number of effectively parallel threads of execution will thus necessarily translate to a lower bound on the required bandwidth from system services providing access to shared physical resources. Since the energy and memory walls apply to system services as they do to applications, the only way forward to reduce contention is parallelism, i.e. multiply physical resources (memories, I/O connections, accelerators) and bound the visibility of each resource to processing elements that are topologically close. The resulting system structure is a constellation topology where each cluster has a limited amount of processing elements sharing common system services, with little to no visibility to the system services of other clusters. This vision has been reached independently, with the same justification, by the authors of [WA09].

14.5 Process state, termination and reclamation

While working on the implementation of system services as per chapter 5 and section 6.4.3, we struggled with process termination. The problem stems from the semantics of the exit system service and termination signals as triggered by POSIX’s kill, abort or C’s raise: when these events are triggered by a program, the run-time environment must terminate the program and reclaim its resources. Since these events can be triggered by one thread while other threads in the same process are active and running, a mechanism must exist to stop the execution of all threads preemptively and reclaim all memory and other system resources that were allocated so far.

Next to process termination, we also struggled to define process data encapsulation. Process-specific data in C comprises all mutable objects defined by programs and library code in the global scope, the shared heap(s) and thread stack(s). The most often used process-specific data from library code are the standard I/O streams (stdin, stdout, stderr), and the error status code variable (errno). While it is possible to encapsulate these variables behind accessors, as in the BSD code which substitutes all uses of the word “errno” by a call to a function “__error(),” the implementation of these accessors itself must have access to process-specific variable instances. The issue we faced was how to look up which instance to use from a given thread, as the proposed machine interface does not provide the notion of process identity.

To summarize, the need to support process data encapsulation as well as resource reclamation upon program termination places two requirements on partial hardware support for concurrency management, in particular the design from part I:

- mechanisms must be available to bind allocated resources, e.g. memory and I/O channels, to process identities which can be used to look up process-specific data and delimit
the extent of resource reclamation upon termination. Our proposal to introduce virtual resource identifiers in the TLS addressing scheme (cf. section 9.4.2.3) makes a first step in this direction; however, a fully general solution requires an interface to bind thread contexts to process boundaries and communicate automatically process identities to system services upon resource allocation events;

- once process identities are available, mechanisms must be available to both inquire from any thread the identity of the surrounding process, and to asynchronously terminate all concurrent activities belonging to a given process.

14.6 Feedback from education activities

Our framework was further used during various education projects at undergraduate levels. Using our proposed interface language and the separately provided full-system emulation, students have implemented various applications (Conway’s Game of Life [Gar70], sorting algorithms, image processing filters, compression algorithms, signal processing, Scheme interpreter, and others) and demonstrated the effective use of single-core multithreading and multi-core execution to accelerate performance with relatively naive source code. The overall conclusions from this work regarding the evaluation of the architecture are similar to those outlined in chapter 13; however, we gained additional experience from working with students:

- comprehensive support for a large subset of the standard C library and POSIX interfaces is a strong prerequisite for students who have little previous exposure to operating system and language design. Indeed, most students seem to have troubles distinguishing whether program constructs are part of the language, the standard library, or the system interfaces. When the focus of a teaching activity is the introduction of new language features (e.g. concurrency constructs in our case), diluting the students’ attention by forcing them to also deconstruct and re-learn C becomes detrimental to their success;

- similarly, a comprehensive troubleshooting infrastructure is essential to keep teaching activities focused, especially to avoid wasting student time with programming errors in the framework or conceptual issues in the new architecture design. Our choice to mandate serializability in the language semantics (cf. section 6.2.4) helped by allowing students to test and troubleshoot their programs independently from the new architecture. However, student progress was hampered by the lack of run-time program introspection tools to visualize trade-offs related to load balancing and scheduling. They also missed troubleshooting tools to help recognize false assumptions, in particular those about the memory consistency model.

To summarize, exposing a new platform to students during teaching activities was an effective way to exercise the tools and increase their accessibility. However, teaching activities confirm the usefulness of conceptual backward compatibility, such as introduced in chapter 5, since it is educationally difficult to introduce simultaneously new programming abstractions and a new vision on the system stack.
Summary

Although our work was taking the perspective of existing and legacy operating software, we were able to recognize some general issues relevant not only to the envisioned audience:

- mechanisms for mutual exclusion: this must be addressed by provisioning and advertising dedicated mechanisms in hardware;
- unexpected events and their reporting to programs: this must be addressed by introducing support for asynchronous notifications;
- the cost trade-offs related to the choice of a centralized MMU for address translation: these must be addressed by the system integrator in a dialogue, taking into consideration the environment where the system is to be used;
- performance bottlenecks due to stateful system services: this must be addressed by provisioning sufficient bandwidth to these services and evolve API designs to expose more concurrency;
- process-specific data and resource reclamation upon termination of programs: this must be addressed by introducing support for process identities;
- troubleshooting and inspection facilities must be provided to aid learning audiences in understanding the run-time behavior of their applications.

These issues must be further researched in future work before the design can be advertised as a general-purpose substitute to any of its candidate ecosystems.