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

Poss, R.C.

Link to publication

Citation for published version (APA):
Chapter 10

Concurrency virtualization
—Highlight on specifics, opus 3

Abstract

A key role of abstraction is to hide the finiteness of physical-world resources, in particular through their virtualization. Virtualization in turn requires specific support from the underlying architecture, and thus constitutes an ongoing interaction between the platform provider and operating software provider. In this chapter, we illustrate this interaction for the virtualization of the “family,” a group of cooperatively scheduled logical threads that can be created and joined using single bulk operations.

Contents

10.1 Introduction ................................................................. 164
10.2 Context and related work ............................................... 165
10.3 Controllable automatic serialization ................................. 166
10.4 Limitations and future work .......................................... 171
Summary .................................................................................. 173
10.1 Introduction

As noted in [Day11], a duality exists between specialization and generalization when constructing abstractions. “Specialization” refers to language restrictions, compile-time (static) solutions and the exploitation of machine-specific facilities—in the interest of efficiency. “Generalization” refers to general language constructs, run-time (dynamic) solutions and machine-independent language design—in the interest of correctness and reliability. When catering to operating software providers, the hardware architect must recognize this duality and enter a dialogue at both levels.

In particular, general abstractions often hide implementation-specific resource limits, to ensure portability and forward compatibility of software. This hiding is achieved by introducing “virtual” resources, which appear unbounded in the abstract semantics of programming languages. Resource virtualization thus achieves a separation of concerns between software specification and its realization at run-time where resource constraints apply. To implement resource virtualization, platform providers and operating software providers must co-design virtualization mechanisms. For example, the virtualization of variable slots in programming languages is typically supported by introducing relative addressing in processors and offset tables in compilers. Meanwhile, the need for specialization requires that users of abstractions can control resources explicitly when so desired [Kic91]. Any virtualization mechanism should thus provide and document optional control to disable or bypass virtualization.

In the context of general-purpose systems, the need for resource virtualization has traditionally been offset by ensuring that most resource abstractions available in programming languages could be expressed in terms of data structures in memory: once multiple resources can be represented by memory, only one virtualization mechanism for memory is needed to virtualize these resources. This is the approach taken e.g. in Unix-like operating systems where inter-process communication channels, files and synchronization devices are all represented by data structures in memory and thus appear unbounded with the backing of virtualized memory. Processes, which are, resource-wise, a combination of virtualized processors and memory, require a separate mechanism to virtualize processors; Unix uses time sharing for this purpose.

When innovating in processor chip design, it is thus necessary to interact with the operating software providers to define virtualization mechanisms for all the resource types introduced. As the Unix experience illustrates, a potential strategy is to represent new resource types in terms of data structures in virtual memory.

10.1.1 Example new resources to virtualize: concurrency contexts

As we explained in part I, the proposed architecture design introduces “execution contexts” for concurrent tasks, dedicated to optimize the management of concurrency throughout the system stack. The architecture design introduces three types of resources: one is the logical thread of execution, another is the bulk creation context to prepare the automatic creation of multiple logical threads, and the last is the bulk synchronizer to synchronize on the termination of multiple logical threads at once.

As explained in section 3.3.1, the proposed machine interface already provides dedicated support to virtualize logical threads: it states that an arbitrarily large number of logical threads are automatically scheduled over the physical thread contexts. Thread virtualization is thus achieved in hardware.
Meanwhile, our proposed abstract machine (chapter 7) defines primitives to create and synchronize on concurrent tasks hierarchically as abstract families, at run-time. Its semantics make an assumption of unlimited hardware resources, by implying that creating logical threads and waiting on their termination always succeeds. In other words, our abstract machine communicates the illusion that programs can always create new families, regardless of where in the program, and when during run-time, creation is expressed. This enables resource-agnostic programming where concurrent program blocks can be freely composed without having to choose which computation is defined as a concurrent family and which computation should be defined as a sequential process.

However, the bulk creation contexts and bulk synchronizers have an explicit, named existence in the machine interface, as explained in chapters 3 and 4. Without support for virtualization of the bulk creation context and bulk synchronizers, it is possible to express programs that may define more families during their execution than there are bulk creation contexts available in hardware. For example, divide-and-conquer algorithms exhibit this situation when the depth of the recursion is data-dependent. With a transparent mapping of the input language construct for family creation to machine primitives, progress and termination of otherwise semantically valid deterministic programs would not be guaranteed, since an extra context allocation when all hardware resources are busy would yield a deadlock or failure (section 4.3.1.1).

To resolve this mismatch, we propose in this chapter a controllable mechanism to virtualize the bulk creation contexts and bulk synchronizers. In section 10.2, we start by outlining how concurrency resources are managed and virtualized in related work. We then outline our proposed protocol in section 10.3 and analyze its limitations in section 10.4.

10.2 Context and related work

The example situation we are addressing here is averted in most existing concurrency management environments, in either of two ways:

- in resource-aware concurrent programming environments, the resource limitations of the underlying execution platform are exposed in the programming language semantics: the programmer or automatic program generator must either acknowledge and assume a fixed set of processing units known a priori, or explicitly query at run-time how many processing units are available before starting parallel execution. This situation can be found, for example, with programs that distribute work over parallel “worker” processes implemented with POSIX processes or threads; the application must query the number of actual processors in the system, and possibly also how many POSIX processes/threads are supported by the operating system, to determine how many workers to start. The Apache web server application\footnote{http://httpd.apache.org/} uses this scheme. It can also be found with the MPI environment, where a process needs to query the number of processes involved in a communicator with MPI_Comm_size before spawning sub-processes with MPI_Comm_spawn.

- in virtualized resource-agnostic concurrent programming environments, most implementations support unbounded numbers of virtual concurrency contexts backed up by main system memory: any request to spawn a new parallel thread of execution is matched by the dynamic allocation, in main memory, of the required data structures.
In this situation, the maximum amount of concurrency that can be defined is typically orders of magnitude larger than the number of hardware processing units, ensuring that the limit is never reached in all practical circumstances.

This situation can be found in most existing task-based concurrency interfaces: tasks in .NET’s Thread Parallelism Library (TPL) [Duf09] and Intel’s Threading Building Blocks (TBB) [Rei07], blocks with GCD [App, Sir09], tasks with OpenMP [Ope08], etc.

In both situations, there is no interface mismatch: in the former case, the input concurrency matches the number of execution contexts by design, and in the latter case the number of execution contexts scales automatically with the amount of concurrency defined by the input program.

Arguably, resource-agnostic programming should be the privileged approach for programming current and future multi-core systems [PJ10]. However, virtualization of execution contexts, where there can be more execution contexts defined than there are processors available, often incurs implementation complexity and run-time overheads: unless some restriction is made on scheduling, space must be allocated from memory to save the state of tasks at every context switch. This incurs contention on memory allocators and extra traffic on the memory network.

An example restriction that alleviates the need for saving the state of all tasks to memory is so-called declarative, resource-agnostic concurrent programming environments. In these, programs declare concurrent operations to perform without indicating whether, where or how to perform them, and the language compiler and underlying platform then cooperate to determine the cheapest and most efficient execution strategy. The specific advantage of declarative concurrency constructs in the associated languages is that any excess concurrency can be serialized cooperatively, as a loop or recursion, automatically within single execution contexts without changes in semantics. As such it constitutes a form of cooperative virtualization.

This is the approach exploited by e.g. the automatic folding of concurrent spawns in the parent task in Cilk [BJK+95], the aggregation of multiple iterations of loops as single tasks in OpenMP [Ope08], the various aggregation strategies of the SAC2C compiler of Single-Assignment C [GS06], the automatic aggregation of parallel algorithms via “partitioners” by Intel’s TBB run-time system [Int11], the automatic serialization of data parallel algorithms by the Chapel compiler and run-time system [Cra11].

### 10.3 Controllable automatic serialization

In this chapter, we propose a possible implementation of declarative concurrency for the proposed target architecture. While doing so, we remove partly from the language semantics the transparency between the concurrency expressed in the program and actual parallelism of execution on the platform at run-time. Our strategy is to delay the choice between concurrent execution, via hardware families, and sequential execution within the context of an existing active thread, via serialization, until run-time when the actual availability of execution units is known. Meanwhile, we reserve the opportunity to control the virtualization via optional language constructs. We thus propose two interfaces to control concurrency:

- when the implicit form of concurrency constructs is used in programs, the automatic scheme is used and run-time resource availability is used to choose between concurrent and serial execution;
10.3. CONTROLLABLE AUTOMATIC SERIALIZATION

- additionally, a programmer, code generator or run-time system implementer can add *explicit specifiers* to the constructs to either force concurrency creation or force sequentialization in a specific syntactic locus.

In both cases, the semantics presented in chapter 7 apply to describe the functional semantics of programs. With the implicit form, progress and completion of workloads are guaranteed by reusing existing contexts for any excess concurrency. With the explicit form, progress and completion are only guaranteed if the entity that wrote the program (programmer or code generator) collaborates with a resource management service on the system to not require more contexts than are available in the actual platform.

10.3.1 Implementation

Prior to the introduction of “soft failure reporting” in the allocation of bulk creation contexts (cf. section 4.3.1.1), we attempted to expose *resource counters* in the hardware that would reflect the occupancy of the hardware structures. We could make these counters visible either via special memory addresses or dedicated instructions in the ISA. However, we found that any counter-based scheme which does not also adapt the family allocation mechanism is flawed, because the fine-grained concurrency in the system makes the value of such counters essentially *inaccurate*: there can be family allocations, or de-allocation events, between a point a counter is sampled and the point the value is acted upon. Generally, any feedback mechanism must ensure that the test on availability and the allocation of a processing resource if there is one available are performed *atomically*.

Instead we co-designed “soft failure reporting” with the architecture implementers. With this feature the “allocate” primitive returns a special “invalid” value upon failure to allocate a bulk creation context. We further introduced a parameter to the operation, to select whether failure allocations are reported to the software (“soft failure”), or whether allocation should wait indefinitely until a context becomes available (“suspending”). The latter option allows to implement mandatory concurrency, or mandatory delegation of an activity such as needed for system services (section 5.5.1), without using busy waiting loops on allocation failures.

Once the feedback mechanism becomes available in the target implementation, we can use it during code generation as follows: whenever execution reaches a point where a concurrent family of threads is *declared* in the input program, the executed code first attempts the context allocation\(^2\). Then, if the result indicates a failure, the code branches to the serialized version of the declared work. Otherwise the declared work is created as a concurrent family of logical threads.

To illustrate this scheme, we consider the example code from listing H.21, which defines a thread program “innerprod” and uses it in a “kernel” function. With our strategy enabled, the translation stage before code generation produces the code given in listing 10.1, which in turn causes code generation and post-processing to produce the assembly code given in listing 10.2 for the SPARC-based target ISA.

This example allows us to highlight the following:

- the scheme requires duplication of the code of thread programs: one version must be generated suitable for creation as a thread (listing 10.1, line 5), and another suitable for a procedure call in C (line 6); furthermore, both must be declared before they are defined, in case the input program uses recursion;

\(^2\)This causes the hardware allocator to attempt an allocation of the desired number of contexts, or any smaller size down to a single context, or fail if no context is available whatsoever.
### Listing 10.1: Automatic serialization code for listing H.21 (simplified).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void _mts_innerprod(void);
void _seq_innerprod(long _I, int * _mtp_a, int * _mtp_b, int * _mtp_sum);
extern void * innerprod[2];

void _mts_innerprod(void) {
    /* thread program */
}
void _seq_innerprod(long _I, int* _mtp_a, int* _mtp_b, int* _mtp_sum) {
    /* plain C function */
}
void * innerprod[2] =
    { &_mts_innerprod, &_seq_innerprod };

int A[100];
int B[100];

sl_def(kernel, sl_shparm(int, res)) {
    long _fid, _start, _limit, _step, _block, _idx;
    int * _arg1, * _arg2, _arg_sum;

    /* common initializers from program source: */
    _start = 0, _limit = 100, _step = 1, _block = 0;
    _arg1 = A, _arg2 = B, _arg_sum = 0;

    /* allocation test: */
    asm volatile("allocate\%0":=r"(_fid)"
        if (-1 == _fid)
            goto _seq;
    
    /* here configure and create as family of threads,
       using context _fid,
       range (_start, _limit, _step),
       window size _block,
       source values _arg1, _arg2, _arg_sum,
       thread program _mts_innerprod,
       then synchronize and update _arg_sum */
    goto _end;

    _seq:
    if (_step > 0)
        for (_idx = _start; _idx < _limit; _idx += _step)
            _seq_innerprod(_idx, _arg1, _arg2, &_arg_sum);
    else
        for (_idx = _start; _idx > _limit; _idx += _step)
            _seq_innerprod(_idx, _arg1, _arg2, &_arg_sum);

    _end:
    _seq_end(res, 0);
}
```
10.3. CONTROLLABLE AUTOMATIC SERIALIZATION

Listing 10.2: Generated assembly code for listing 10.1.

```assembly
.global _mts_kernel
.type _mts_kernel, #function
.registers 0 1 6 0 0 0

_allocate %tl3 !. ‘try or fail’
cmp %tl3, -1; swch !. success?
be .LL13 !. no: branch below
nop
sethi %hi(A), %tl0 !+ load channel source
or %tl0, %lo(A), %tl0 !+ load channel source
sethi %hi(B), %tl1 !+ load channel source
or %tl1, %lo(B), %tl1 !+ load channel source
mov 0, %tl2 !+ load channel source
setlimit %tl3, 99 !+ configure limit
sethi %hi(_mts_innerprod), %tl5 !+ load PC
or %tl5, %lo(_mts_innerprod), %tl5
setthread %tl3, %tl5 !+ configure PC
create %tl3, %tl3; swch !+ create family
mov %tl3, %tl3; swch !+ wait on termination

.LL14:
mov %tl2, %ts0 !. propagate result
end !. end thread

.LL13:
sethi %hi(B), %tl4 !- load argument
sethi %hi(A), %tl5 !- load argument
mov 0, %tl10 !- init counter
mov 0, %tl12 !- load argument
or %tl14, %lo(B), %tl14 !- load argument
or %tl15, %lo(A), %tl15 !- load argument

.LL15:
ld [%tl14+%tl10], %tl13 ! inlined load
ld [%tl15+%tl10], %tl11 ! inlined load
add %tl10, 4, %tl10 !- increment counter
smul %tl3, %tl11, %tl11; swch ! inlined mul
cmp %tl10, 400 !- test counter
add %tl12, %tl11, %tl12; swch ! inlined reduction
bne .LL15 !- next iteration
nop
b,a .LL14 !- back to common end
nop
.size _mts_kernel, .- _mts_kernel
```

Listing 10.2: Generated assembly code for listing 10.1.
• once the definition of the C function is in scope, the inlining capabilities of the code generator can operate as usual; this is demonstrated in the example where the code generator inlines the kernel loop entirely (listing 10.2, line 31);
• although there are two loops to cater for both increasing and decreasing index sequences, usually constant propagation of the step value by the code generator will ensure that only one branch is compiled, as demonstrated in the example;
• the overhead of the scheme compared to a pure sequential code using a loop is 3 machine instructions: one use of family allocation (listing 10.2, line 6) and a conditional branch (lines 7 and 8);
• the overhead of the scheme compared to a transparent use of the concurrency primitives of the underlying architecture is 2 machine instructions, i.e. the conditional branch;
• although the two versions can be separately identified by name mangling\footnote{e.g. “\_mts\_innerprod” vs. “\_seq\_innerprod”}, we also need to generate a single function descriptor (listing 10.1, line 8), defined as an array of pointers to the two codes and with the original name “innerprod,” to allow a single address for the function to be taken and stored in C data structures, as required for system code;
• although an allocation failure causes a serialized version of the family-defining loop to be used, this does not imply that the family is fully serialized: if the thread program of the serialized family uses further family creations, these will attempt to allocate a concurrent context again. This allows parallelism to be re-gained dynamically once some contexts become available.

10.3.2 Impact on the language semantics

Thanks to the requirement of serializability introduced early on (section 6.2.4), the base language semantics as of chapter 7 and Appendix I did not require any change with the introduction of this automated resource scheduling scheme in the implicit form of the concurrency constructs. In particular, all our definitions about scheduling, ordering of side effects, etc. discussed in chapter 7 apply to the new operational semantics of our implementation just as well as they did without the scheme implemented.

However, as described in section 10.3, we also found necessary to cater for the need of system code by providing an explicit control over this scheme. We did this by extending the syntax form of create specifiers (Appendix I.5.8.1):

\texttt{create-specifier:}

\begin{verbatim}
... 
sl\_\forceqseq
sl\_\forceqwait
\end{verbatim}

With this syntax extension, it becomes possible to use the words “sl\_\forceqseq” or “sl\_\forceqwait” as the 6th positional parameter of the “sl\_create” construct. The code translator can then determine which allocation mode to use, respectively “wait and succeed” or “try or fail,” and also avoid emitting the version of the creation code that is left unused (respectively the serialized and non-serialized part). See also side note 10.1.

From a semantics perspective, these specifiers do not modify the functional behavior of the program; however we intentionally choose to not advertise their existence in our
Side note 10.1: Static vs. dynamic allocation failure management.

The syntax-based discrimination is static: the specifiers are lexical keywords and it is their presence or absence in the syntax that determines the choice at compile-time.

We chose a static scheme over a dynamic scheme, i.e. where the run-time value of an expression would determine the behavior, because we found that in all situations where control is desired, we know statically which implementation should be used. If future work shows that a dynamic choice is required instead, we estimate that the additional implementation work required to add a run-time conditional in the generated code is minimal; however we highlight that a run-time conditional would prevent the static elision of whichever version is not used from the generated code.

<table>
<thead>
<tr>
<th>Array Positions</th>
<th>Clock Cycles (x1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 10 15 20 25 30 35 40 45</td>
<td>(Sequential)</td>
</tr>
<tr>
<td>0 5 10 15 20 25 30 35 40 45</td>
<td>(1 family context)</td>
</tr>
<tr>
<td>0 5 10 15 20 25 30 35 40 45</td>
<td>(32 family contexts)</td>
</tr>
</tbody>
</table>

(a) Sequential  
(b) 1 family context  
(c) 32 family contexts

Figure 10.1: Execution of QuickSort on 1 core.

These diagrams represent memory activity over time. The horizontal axis represents time; the vertical axis represents cells in the array being sorted. A dot in the diagram represents a memory read or write operation. In a sequential execution, only one branch of the recursive divide-and-conquer step is executed at a time (e.g. figs. 10.1a and 10.1b); with concurrency, multiple branches are executed simultaneously (e.g. fig. 10.1c).

10.3.3 Run-time behavior

In Appendix J we detail the behavior of two naive recursive QuickSort implementations with our strategy enabled.

As expected, the use of our concurrency constructs adds overhead compared to a pure sequential version if the program is forced to run as a single thread, because the hardware must stall the execution at every attempt to allocate a bulk creation context until the allocation unit reports a failure. However, as soon as more than one context is available in hardware, it is automatically exploited by the program and the resulting multithreaded execution provides latency tolerance and speedup, even on one core (fig. 10.1).

Interestingly, we found that it is not very beneficial to instrument the naive version to only use concurrency when the amount of work at a given recursion level, here the size of the sub-array to sort, is larger than a threshold (fig. 10.2). Indeed, since the cost of testing the sub-problem size is comparable to the cost of trying an allocation, any threshold-based alteration to the scheme amounts to trading one time overhead with the same.

10.4 Limitations and future work

As seen above, our scheme incurs a check overhead, with longer execution times than a pure sequential program, if concurrent code executes in a mostly sequential environment.
Furthermore, this overhead cannot be trivially avoided by means of a threshold on the problem size, because simple conditionals have a comparable cost to a failed allocation.

We generalize by stating that a program containing dynamically heterogeneous concurrency should only be expressed using concurrency constructs in the source code if a reasonable confidence exists that multiple family contexts will be available at run-time to provide multithreading overlap and compensate for the overheads. This confidence can be provided by an on-chip resource manager that partitions the chip between application components (cf. chapter 11). Otherwise, the concurrency management overheads contribute to the critical path through the execution, as highlighted in [FLR98].

We suggest that the scheme could be extended by duplicating the entire recursion as a purely sequential program, with a unique start-up check in the algorithm entry point on whether the underlying execution resource is sequential or not. In the sequential case, the sequential code is used and this overhead could be avoided. This possible extension constitutes future work.

There is another limitation that can be observed when running families defining many logical threads, for example one thread per item in an array. If an algorithm exhibits both dynamically heterogeneous concurrency, e.g., data-dependent divide-and-conquer, and large families, it becomes possible that the serialization of a family causes a load imbalance: once a leaf family starts to run as a loop, it will execute entirely as a loop, even if concurrency contexts become available during the loop execution. We suggest to extend our scheme to try and allocate contexts within the loop body, and switch back to parallel execution when a context becomes available. To avoid a detrimental effect on the performance of inner loops, these attempts should be performed by an outer loop with a configurable coarser step than the inner loop. This should be explored in future work.

Furthermore, while working on TLS (chapter 9), we discovered another limitation to this scheme: the serialization of a family of threads as a loop also merges any thread-local storage requirements, e.g. minimum stack frame sizes, of the serialized family into the thread where the serialization occurs. In other words, in any situation where serialization may occur, the execution environment must provision extended thread-local storage to each serializing thread context. With recursion, the thread-local storage of some threads may become arbitrarily large during execution. This in turn requires an efficient, low-latency dynamic memory allocation scheme with extremely low contention to keep all threads on each core active [Gru94]. Concurrent dynamic memory management is a known active field of research, and we expect any future result in that direction to be relevant to the architecture considered.
10.4. LIMITATIONS AND FUTURE WORK

Finally, we highlight that the virtualization scheme described in this chapter is cooperative. In particular, it requires that the programming language semantics do not expose mandatory parallelism, i.e. the ability for a program description to require from the platform that two activities must be carried out in parallel (e.g. so that they can communicate bidirectionally). This requirement holds for languages like our proposed interface from chapters 6 and 7, where the main constructs for concurrency are declarative. However, as we suggested in section 6.3.6 and detail further in chapter 12, other programming interfaces may be devised for the platform. For example an implementation of Unix would require another form of concurrency virtualization that guarantees fair scheduling between independent processes. If or when this happens, and an alternate interface offers mandatory parallelism in its semantics, other virtualization mechanisms will be required.

Summary

- In this chapter, we have highlighted the dialogue between the hardware architect and the operating software provider to design virtualization mechanisms that abstract away the finiteness of physical resources. This dialogue must be engaged each time a new resource type is added to a platform design.
- We have illustrated this dialogue with the example of concurrency resources in the proposed architecture, which are finite on any given implementation and are desirably unbounded in abstract models. We have addressed this virtualization need by introducing declarative concurrency constructs in the interface language. We have also proposed an implementation of these constructs that automatically and cooperatively sequentializes units of work at runtime when concurrency resources in the underlying platform become exhausted. We have isolated a need to manage the mapping of workloads to resources in future work. We have also highlighted that declarative concurrency may not be suitable for other programming abstractions, where different virtualization mechanisms may be required instead.