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

Poss, R.C.

Citation for published version (APA):
Chapter 8
Configuration of the visible synchronizer window
—Highlight on specifics, opus 1

Abstract

Some architectural innovations *may be eventually invisible to application developers* and thus not part of the “advertised” main features of a design. Yet they may significantly impact issues under control of operating software providers, and thus must be fully described for this audience. The proposed machine interface from chapter 4 illustrates this. In traditional register machines, the number of register names available for use in programs is fixed by the ISA definition. In our case, the code generator that produces the machine code can *choose* the number of register names available for use: as per sections 3.3.3 and 4.3.3, the hardware dynamically maps register names to physical synchronizers. The set of mapped synchronizers constitutes a “visible window” from the perspective of individual threads. A lower requirement on the number of available register names enables more compact use of the synchronizing storage, potentially more thread contexts active, and thus potentially better opportunities for latency tolerance. In this chapter, we describe to operating software providers how a code generator can configure the layout of the visible window.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>138</td>
</tr>
<tr>
<td>8.2</td>
<td>Formalism</td>
<td>138</td>
</tr>
<tr>
<td>8.3</td>
<td>General constraints</td>
<td>139</td>
</tr>
<tr>
<td>8.4</td>
<td>Degree of freedom from register allocation</td>
<td>140</td>
</tr>
<tr>
<td>8.5</td>
<td>Strategies for synchronizer mapping</td>
<td>141</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td>146</td>
</tr>
</tbody>
</table>
8.1 Introduction

The machine interface in chapter 4 allows software to configure a separate register window layout for each thread program. Specifically, a code generator must produce a triplet of values $G$, $S$, $L$ in the machine code for each thread program. These specify the number of synchronizers that must be made available by the architecture at run-time before execution of the thread program starts; respectively, the number of “global” synchronizers visible from all thread contexts, the number of “shared” synchronizers shared by adjacent contexts, and the number of “local” synchronizers private to each context (cf. figs. C.1 and C.2 for an example). These values are constrained only by the equation $G + 2S + L \leq M$, where $M$ is the total number of register names available for use by programs in the ISA, commonly 31.

The question thus arises of how to determine these values.

We have explored multiple strategies. In this process, we determined the general upper and lower bounds on these values, as well as the design spectrum of possible solutions, which we describe. Our implementation uses the outcome of our findings, described below. We distinguish in our text further between “synchronizers,” which are the physical units of storage, and “register names,” which are the addresses used as operands in machine instructions.

8.2 Formalism

We start by formalizing as follows:

- we define the set $\mathcal{P}$ of all machine representations for thread programs on the architecture, and
- the set $\mathcal{F}$ of all machine representations for regular functions (which can be called from a thread, not executed as a family of threads).

Then for any $P \in \mathcal{P}$ we can define:

- $L(P)$, the number of register names reserved as local synchronizers in the declared interface (cf. section 4.3.3);
- $G(P)$, the number of register names used for incoming “global” channel endpoints (cf. section 4.3.3.2);
- $S(P)$, the number of register names used for outgoing “shared” channel endpoints (cf. section 4.3.3.3);
- $R(P)$, the highest register name actually used as local synchronizer in $P$ ($R \leq L$);
- $D_c(P) \subseteq \mathcal{P}$, the set of thread programs that are used as the target of family creation in $P$;
- $D_f(P) \subseteq \mathcal{F}$, the set of functions called directly from $P$;

and for any $F \in \mathcal{F}$ we can define:

- $R(F)$, the highest register name actually used as local synchronizer in $F$;
- $D_c(F) \subseteq \mathcal{P}$, the set of thread programs that are used as the target of family creation in $F$;
- $D_f(F) \subseteq \mathcal{F}$, the set of functions called directly from $F$. 

8.3 General constraints

• We also define $D_f^*(x) \subset \mathcal{F}$, the set of all functions called transitively from $x$ (from either $\mathcal{P}$ or $\mathcal{F}$), defined by:

\[
\begin{align*}
D_f^0(x) &= D_f(x) \\
D_f^{n+1}(x) &= \{D_f(y) \parallel y \in D_f^n(x)\} \\
D_f^*(x) &= \bigcup_{n \to \infty} D_f^n(x)
\end{align*}
\]

(This set is finite on any actual implementation of the architecture due to the finiteness of the address space)

8.3 General constraints

□ From the definitions above, we are able to formulate the following constraints on $G$, $S$ and $L$:

1. outer interface fit:

\[
\forall P \in \mathcal{P} \quad G(P) + 2S(P) + L(P) \leq M
\]

This denotes that the window specification must fit the maximum size in the substrate ISA. The factor 2 for $S$ comes from the fact that there are two sets of “shared” synchronizers: those shared with the previous context and those shared with the next context (cf. figs. C.1 and C.2).

2. required lower bound for calls:

\[
\forall F \in \mathcal{F} \quad \left( \max_{F' \in D_f^*(F)} R(F') \right) \leq R(F)
\]

\[
\forall P \in \mathcal{P} \quad \left( \max_{F \in D_f^*(P)} R(F) \right) \leq R(P) \leq L(P)
\]

This denotes that must be enough private synchronizers to allow the code of any directly and indirectly called functions to run within the same context.

3. required lower bound for creations, in interfaces with “hanging” synchronizers (section 4.3.3.3):

\[
\forall P \in \mathcal{P} \quad \left( \max_{P' \in D_c(P)} (G(P') + S(P') + 1) \right) \leq R(P) \leq L(P)
\]

\[
\forall F \in \mathcal{F} \quad \left( \max_{P' \in D_c(F)} (G(P') + S(P') + 1) \right) \leq R(F)
\]

This denotes that there must be enough private synchronizers to serve as channel endpoints for all the created families.
CHAPTER 8. VISIBLE SYNCHRONIZER WINDOWS

8.4 Degree of freedom from register allocation

We focus here on the distinction between the number of available register names, which is the value declared with $L$, and the number of effectively used register names $R$ in a given machine representation.

To understand their relationship, we must consider how register allocation works in a compiler. To summarize, a compiler considers the maximum number of simultaneously active local (C’s “automatic”) variables in a function body, together with intermediate results in expressions, and deduces the required number of frame slots $N$ for a function or thread program body. $N$ is independent of the target architecture configuration. Then the variables that are most used are mapped onto $A$ available register names, where $A$ is typically configurable. We illustrate the functional nature of register allocation in fig. 8.1.

4. required lower bound for creations, in interfaces with “separated” synchronizers (section 4.3.3.3):

\[
\forall P \in \mathcal{P} \quad \begin{cases} 
0 \leq R(P) \leq L(P) & \text{if } D_e(P) = \emptyset \\
1 \leq R(P) \leq L(P) & \text{otherwise.}
\end{cases}
\]

\[
\forall F \in \mathcal{F} \quad \begin{cases} 
0 \leq R(F) & \text{if } D_e(F) = \emptyset \\
1 \leq R(F) & \text{otherwise.}
\end{cases}
\]
If \( N \leq A \), then \( R = N \leq A \) register names are used. If \( N \geq A \), then \( R = A \) register names are used and the code generator inserts extra instructions to save and restore value slots to and from memory. We illustrate this in fig. 8.2, where a single C function performing arithmetic on an array is processed through various values of \( A \). Traditionally, \( A \) is fixed by subtracting a number of “reserved” names (e.g. stack pointer) from \( M \). It then becomes an architectural constant valid for all code generators on that architecture. For our architecture, we can choose both \( L \) and \( A \), with the constraint that \( A \leq L \leq M - G - 2S \).

In short, next to \( G \), \( S \) and \( L \) there is an additional degree of freedom \( A \) which is an input to code generation.

### 8.5 Strategies for synchronizer mapping

We considered only strategies for synchronizer mapping that account for separate compilation, that is, a given thread program or C function may be compiled without knowing the definition of all other thread programs or functions that it composes through family creation or C function calls. We acknowledge that there exist other strategies when all the program source is visible during code generation, but we decided early on to focus first on providing a platform where separate compilation is possible.

With separate compilation, any strategy to compute a synchronizer mapping for a function needs to account for the possibility that the resulting machine code may be composed in a yet unknown program context. This requires any strategy to select \( G \) and \( S \) using only the prototype declaration of the thread program being considered, since this is the only information available from another translation unit during separate compilation. In particular, it is not possible to choose \( G \) and \( S \) as a function of \( L \), since \( L \) will not be known in another translation unit while \( G \) and \( S \) will be required to generate code for family creations.

Furthermore, to ensure compilation soundness, the strategy should not cause a valid program composition in the semantics of the source language to have no valid representation in the machine code.

We considered first the numbers \( G_s \) and \( S_s \) of “global” and “shared” channel endpoints defined in the program source. Then we considered the simplest possible strategy, i.e. use \( G = G_s \), \( S = S_s \), then choose \( A = L = M - G - 2S \). There are three problems with this strategy. To understand why, we first distinguish fitting programs where \( G_s + 2S_s \leq M \) and overflowing programs where \( G_s + 2S_s > M \). Then:

1. The strategy is “synchronizer-greedy,” since a simple thread program which may only use a few synchronizers and channel endpoints will still end up reserving all \( M \) register names from the architecture.
2. The strategy is not sound. There are two situations:
   - with “hanging” synchronizer mappings (section 4.3.3.3), once a fitting program \( P_1 \) is compiled, it becomes impossible to separately compile another program \( P_2 \) which creates a family running \( P_1 \) if \( G(P_1) + S(P_1) + 1 \geq M - G(P_2) - 2S(P_2) \), even though \( P_2 \) on its own may be fitting. In other words, this strategy establishes that whether a thread program can be used to create a family is a function of the interface of the creating thread.
   - with “separated” synchronizer mappings, once a fitting C function \( F \) is compiled \((R(F) \leq M)\), it may be impossible to separately compile a thread program \( P \) which calls \( F \) if \( R(F) \geq M - G(P) - 2S(P) \), even though \( P \) on its own may
be fitting. (The difference with the previous argument is that it is now plain C
function calls, instead of thread creations, that defeat soundness.)

3. The strategy requires the code generator to reject overflowing programs. This is unsat-
s satisfactory, because it requires the programmer to know \( M \) to decide whether a program
is valid, and it establishes a lower bound on \( M \) for future architectural changes as soon
as a program is written.

We addressed these problems in turn, as described below, to establish the strategy we
finally used in our implementation.

8.5.1 Minimizing the number of local synchronizers

To solve the first problem, we can choose \( A = M - G - 2S \) first, let register allocation run
and use the resulting \( R \) as a value for \( L \) (this is possible since \( R \leq A \)). The alternative is
choosing \( A < M - G - 2S \), to reduce the mandatory synchronizer cost of the thread program
even further. However, as seen in the previous section, when \( A \) becomes smaller, the code
generator must introduce extra memory load and store instructions for spills. This means
that using \( A \) to control the synchronizer cost is effectively a trade-off between this cost and
memory access penalties. To distinguish the benefits and drawbacks, we must therefore look
more closely at the hardware implementation:

- in an interleaved multithreaded execution (one pipeline, one synchronizer store), adding
memory operations is always a penalty, even if all memory latencies can overlap with
useful computations, because the total number of executed instructions increases. In
this situation, the largest value \( A = M - G - 2S \) should be used to minimize the number
of memory accesses in all cases.
- in a simultaneous multithreaded execution (multiple pipelines sharing one synchronizer
store), \( A \) should be reduced so that there can be enough threads to keep all functional
units busy and also tolerate the latencies of the added memory latencies.

Since the implementations we used have one in-order pipeline per synchronizer store, we
choose \( A = M - G - 2S \) accordingly.

8.5.2 Ensuring sound composition

8.5.2.1 Case with “hanging” synchronizer mappings

We consider first “hanging” synchronizer mappings as introduced in section 4.3.3.3. In this
case, any upper bound for \( G \) and \( S \) for a given program \( P \) becomes a lower bound on the
choice of \( L \) for any other program that wishes to create a family running \( P \). A lower bound
on \( L \) is in turn an upper bound on \( G \) and \( S \) in that other thread. Since any solution must
ensure compatible values of \( G \), \( S \) and \( L \) for family compositions across separate translation
units, we must establish a fixed point for the upper bound on \( G \) and \( S \), and the lower bound
on \( L \). Let us call this fixed point \( X \). By definition, \( \forall P \in \mathcal{P} \quad G(P) + 2S(P) \leq X \) and
\( G(P) + S(P) + 1 \leq L(P) \leq M - X \).

Furthermore, \( X \) must be independent from specific program sources, since \( X \) must be
known by convention across all separate compilations. This means that \( X \) can be chosen
arbitrarily. Once the fixed point is chosen, we can deduce from \( X \) the upper bounds on \( G \),
\( S \) and \( L \) as per the definition above.
8.5. STRATEGIES FOR SYNCHRONIZER MAPPING

![Diagram showing maximum allowable values of G, S, L for various values of X, with M = 31.](image)

Figure 8.3: Maximum allowable values of $G$, $S$, $L$ for various values of $X$, with $M = 31$.

Configurations suitable for “hanging” synchronizer mappings, as per section 4.3.3.3.

Since register allocation benefits from large values of $L$, and inter-thread communication benefits from large values of $G$ and $S$ (more asynchrony), we are thus interested to choose $X$ in a way that maximizes $L$, $G$ and $S$. To clarify, we plotted in fig. 8.3 the upper bounds for all values of $X$ with $M = 31$. As can be seen in this diagram, all values of $X$ below 15 enable more communication endpoints at the expense of local synchronizers. Values between 15 and 20 allow more “shared” channel endpoints at the expense of both “global” channel endpoints and local synchronizers. Values above 20 restrict all three values further.

The optimization of $X$ would require empirical results across a set of applications. Indeed, a low value of $X$ can require inter-thread communication to occur more often via memory instead of the architecture’s dedicated channels, while a high value of $X$ can require more spills to memory due to a smaller number of private synchronizers. The best trade-off for a given application depends on the source code of the entire application and how program components are composed together; the best trade-off for an entire application domain requires empirical evaluation across the entire domain.

Without considering a specific application domain, we made $X$ configurable and set it by default to $X = 12$. We motivated this default value as follows. First we observed that a prospective user of our technology, the SAC2C compiler for Single-Assignment C [GS06], was structured to make heavy use of common loop variables, which would be translated to “global” channel declarations in our interface language. This suggested maximizing the upper bound on $G$, and thus choosing the largest possible value of $X$ while keeping $X \leq 15$. Then we observed that the calling conventions for procedure calls on existing general-purpose processors with RISC ISAs often choose to use 4 to 8 register names to argument passing (table 8.1), plus 6 to 12 “scratch” caller-save register names for temporary variables. Then we deduced from this observation that the designers of these calling conventions must have empirical knowledge that this is an optimal trade-off to support C functions and calls in general-purpose workloads. So we adopted this requirement in our setting, and we deduced the constraint $L \geq 18$, hence $X = 12$.

Once $X$ is set, the resulting code generation strategy can be summarized as follows: first set $G$ and $S$ from $G_s$ and $S_s$ while ensuring $G + 2S \leq X$; then run code generation with $A = X$, and finally set $L = R$ after code generation. This is the implementation we describe in Appendix H. The procedure calling convention within threads is left unchanged from the substrate ISA.
CHAPTER 8. VISIBLE SYNCHRONIZER WINDOWS

<table>
<thead>
<tr>
<th>ISA</th>
<th>ABI</th>
<th>Integer arg. regs.</th>
<th>FP arg. regs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC</td>
<td>(all)</td>
<td>3-10</td>
<td>33-40</td>
</tr>
<tr>
<td>Alpha</td>
<td>(all)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>SPARC32</td>
<td>System V, Sun</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>SPARC64</td>
<td>Sun</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>MIPS32/64</td>
<td>“old”</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>MIPS32/64</td>
<td>“new”</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>ARM</td>
<td>Standard [ARM09]</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>x86-64</td>
<td>Unix</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>x86-64</td>
<td>Microsoft</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 8.1: Calling conventions on existing general-purpose processors.

Information extracted from the GNU CC target machine configurations.

Figure 8.4: Maximum allowable values of $G$, $S$, $L$ for various values of $X$, with $M = 31$.

Configurations suitable for “separated” synchronizer mappings, as per section 4.3.3.3.

8.5.2.2 Case with “separated” synchronizers

We consider here “separated” synchronizer mappings as per section 4.3.3.3. Any upper bound for $A$ (or $R$) for a given C function $F$ becomes a lower bound on $L$ for a program $P$ that wishes to call $F$. A lower bound on $L$ is in turn an upper bound on $G$ and $S$ in $P$. Since any solution must ensure compatible values on $G$, $S$, $L$ and $R$ for function calls across separate translation units, we must establish a fixed point for the upper bounds on $G$ and $S$, and the lower bound on $L$. Let us call this fixed point $X$. By definition, $\forall P \in \mathcal{P} \quad G(P) + 2S(P) \leq X$ and $1 \leq L(P) \leq M - X$.

As described before, $X$ must be known by convention, and once known it can be used to derive $L$, $G$ and $S$ in thread programs, and $A$ (or $R$) in C functions. As before, we are interested to choose $X$ in a way that maximizes $L$, $G$, $S$, and $R$ for C functions. With the new constraint, we plotted in fig. 8.4 the upper bound for all values of $X$ with $M = 31$. As can be seen in this diagram, increasing $X$ allows for more channel endpoints as the expense of local registers.

Again, the optimization of $X$ requires empirical data across a target application domain. The trade-offs remain unchanged, as well as the methodology. We thus felt justified to reuse the same semi-arbitrary choice of $X = 12$ in our setting, for the same reason as above. This allows us to keep the previous decision strategy for $G$, $S$ and $L$ unchanged.
8.5. STRATEGIES FOR SYNCHRONIZER MAPPING

Note that we use here the constraint between $X$, $G$ and $S$ stemming from program composition via C function calls under separate compilation, whereas the constraint we used in section 8.5.2.1 stemmed from composition via thread creations. The reason why we did not use the same constraint in both sections is that the constraint from composition via thread creation is more restricting with “hanging” synchronizer mappings, whereas it is less restricting with “separated” mappings.

8.5.3 Handling overflowing programs

The last problem was dealing with programs where the number of expressed channel endpoints is too large for the code generation constraints. In our case, with $X = 12$, we can use at most 12 hardware endpoints for “global” channels directly, or 6 endpoint pairs for “shared” channels, or any combination where $G + 2S \leq 12$. We could propagate these constants to the input language definition, and thus make programmers (or code generators) aware of this hardware limit and accept the invalidity of overflowing programs as a design choice; however, an additional improvement is possible.

Indeed, we can reuse the strategies from [JR81] to support virtual inter-thread channels on the new architecture. With only one physical “global” channel endpoint, it is possible to emulate a virtually unbounded number of other “global” channel endpoints by storing the source values in memory instead. For example, the program fragment in listing 8.1 can be replaced by listing 8.2 without any changes in semantics. In general, any number of “global” endpoints declared in source can be mapped to $G \geq 1$ endpoints in hardware, where the first $G - 1$ hardware endpoints are mapped directly to program-specified endpoints, and the last
endpoint in hardware is mapped to a pointer to a data structure in memory that contains the
source values for the other program-specified endpoints. We call this method \textit{escaping} the
channel source values into memory, because it trades physical hardware channels for memory
capacity. When we use escaping, no value of \( G_s \) causes the program to be overflowing as
soon as \( G \geq 1 \) is allowed by \( X \). We actually implemented this method in our solution.

The method can be further extended to “shared” channels by reserving a data structure
in memory in each logical thread in a family to hold the source values for the next logical
thread.

However, here a difficulty arises. We cannot translate multiple individual uses of \texttt{sl_setp}
that manipulate different virtual “shared” channel names to multiple uses of \texttt{sl_setp} that
manipulate the same hardware channel endpoint. This is because the machine interface
does not allow us to reset the synchronization state of the channel, i.e. we must assume the
channel only synchronizes once. This requires us to replace all escaped uses of \texttt{sl_setp}
by non-synchronizing memory assignments, and \textit{also} add a single use of \texttt{sl_setp} at the earliest
point of the control flow graph that is guaranteed to run after all the memory assignments.

Unfortunately, this would be a program transformation that requires semantic analysis of
the program source, which is outside of the scope of the framework we discussed in chapter 6.
For this reason, we originally chose to not extend the method to “shared” channels in our
implementation. Then we realized that there exists an alternative, to add the trailing
\texttt{sl_setp} at the end of the control flow graph, i.e. at the end of the function body and at
every use of \texttt{return}. This would be a suitable context-free substitution. The drawback of
this method is that it over-synchronizes by forcing the sequentialization of all participating
threads. If proper virtualization of “shared” channels is desired, this technique could be
added to our implementation at a low cost. The reason why we did not yet implement this
solution is suggested in section 13.8.

Our resulting implementation can be summarized as follows:

- in the language specification (Appendix I, clause I.4.2§2), we guarantee support for
  127\(^1\) “global” channel endpoints and 1 “shared” endpoint pair;
- in the actual implementation, the number of “global” endpoints is effectively bounded
  only by the memory available to the creating thread, and the number of “shared”
  endpoint pairs is indirectly bounded by \( X \), to 6 pairs in our case; however, we do not
  advertise this value so as to enable other implementation-dependent choices for \( X \).

\textbf{Summary}

Variably sized visible windows influence register allocation during code generation. To enable
separate compilation, a static limit must be set on the number of register names available
to register allocation in code generators for local variables, which in turn constrains the
number of register names available to define synchronization channel endpoints.

In this chapter, we have formalized the possible trade-offs in this choice. We then described
how to virtualize an arbitrary number of logical channels defined in programs over a fixed set
of register names in the ISA. The result is a configuration strategy that is mostly invisible
from the perspective of the C programmer or higher-level code generator.

\footnote{For symmetry with the limits on the number of function arguments in [II99, 5.2.4.1§1] and [II11b,
5.2.4.1§1].}