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

Poss, R.C.

Citation for published version (APA):
Chapter 6

First level programming environment —A “quick and dirty operating system” for microthreaded chips

Abstract

In this chapter, we describe how to provide an implementation of C to the new proposed architecture. We use non-standard features from GNU CC in an effective, yet simple way to generate code for the new architecture. We also define a simple input syntax to control the concurrency management primitives of the target machine, which we call “SL.” We then generalize our approach as a compiler combinator which can extend other code generators to recognize our input syntax. Finally, we use our technology to port C library and operating system services to the platform introduced in chapter 5.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Prior work</td>
<td>98</td>
</tr>
<tr>
<td>6.2</td>
<td>Guiding considerations for an interface language</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Proposed code generator</td>
<td>103</td>
</tr>
<tr>
<td>6.4</td>
<td>Operating software services</td>
<td>110</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td>113</td>
</tr>
</tbody>
</table>
CHAPTER 6. PROGRAMMING ENVIRONMENT

6.1 Prior work

- Prior to the work described in this dissertation, some research had been realized to design a C-based interface to the proposed architecture. This work was advertised starting with [Jes06a] and culminating with [Ber10]. At the start of our research, we carried out an extensive analysis of the design and semantics of this proposed interface, which we detail in Appendix G.

To summarize, this language was based on a few language primitives that semi-transparently expose the ISA semantics introduced in chapter 4. However, through analysis we were able to identify the following shortcomings, not described in previous work:

- the proposed design only implements a subset of C, which conflicts with the requirements described below in section 6.2.1;
- it introduces significant complexity in the front-end and middle-end of a C compiler, by requiring obtuse abstract semantics for the handling of synchronization channel endpoints. We describe this in Appendix G.2;
- it cannot be compiled to first generation interfaces (cf. section 4.7), which prevents its applicability to the UTLEON3 platform which we intended to support. We prove this in Appendix G.3.

While we attempted initially to fix the design with minor semantic restrictions to extend its applicability to all our target platforms, we were not able to find a satisfying simple solution to all these shortcomings. We thus chose early to not invest further effort in this direction. Instead, we point out that these shortcomings are a consequence of a research strategy that attempts to design a language prior to and independently from its prospective implementations in a compiler.

6.2 Guiding considerations for an interface language

6.2.1 Choice of C, language subset or superset

The choice of C as a starting point, as opposed to some other language, can be debated. Indeed, C presents the burden that it was designed for mostly sequential computers, and the programmability and generality of the proposed architecture design could be demonstrated using languages designed with concurrency in mind, for example Erlang [AVWW96] or Google’s Go [Goo].

Yet it is customary to implement C on new architectures. This is motivated first by the ubiquity of C implementations that can be customized, the wide audience for the language, and its de facto status as the low-level hardware interface language for the audiences identified in chapter 5 (including its role in implementing portable operating systems). Beyond these practical motivations, we found that there are four fundamental properties of C that are relevant in architecture research:

1. C is not a fixed language defined by specification; it is a family of languages, each defined by the actual capabilities of its implementation, and the intention for interoperability between implementations.
6.2. GUIDING CONSIDERATIONS FOR AN INTERFACE LANGUAGE

Side note 6.1: Contenders to C to pitch new architectural designs.

**LLVM IR/bytecode.** This is a strong contender with C, however its developer communities still lack the multiculturalism and distributedness of existing C communities as of this writing;

**CLI (.NET) / JVM.** These are not multifaceted, not lean, not multicultural, not distributed, and place strong requirements on the underlying platform: mandatory isolation, mandatory preemption, mandatory coherent caches, etc.;

**C++.** Its language design derives from C, and most C++ compilers contain parts of a C compiler as a substrate. The strategy we describe in the rest of this chapter rely on the parts of a C compiler common with C++ compilers, and could thus be applied to C++ with no changes.

---

2. Those shared semantics of most C implementations that are used as reference by programmers, and thus collectively named “standard” and captured in [II99, III1b], carefully and deliberately avoid making assumptions about features available in the hardware machine underlying any specific C implementation (any exceptions, e.g. required support for preemption in signal, are encapsulated in library services, i.e. outside of the core language).

3. Moreover, the C language is purposefully designed for hardware transparency, that is, the language semantics do not attempt to hide the hardware machine interface.

4. All implementers of C are allowed to extend their implementation with new features not found in other implementations; these can then be publicly scrutinized and discussed by international communities of industrial, academic and private experts. Successful extensions are usually adopted by other implementers and become candidate for future standardization.

It is this combination of well-structured, multifaceted and lean semantics, an implementation-driven community, a tolerance for and promotion of hardware diversity, and a multicultural, distributed meritocratic approach to new developments, not found combined in any other language, that makes C unique and especially desirable for hardware architecture research.

We were asked to comment on the opportunity to support only a restricted subset of the C language in a new implementation. This seems interesting because any specific set of evaluation or benchmark programs only needs the features sufficient to compile them, and restricted language support may reduce the amount of testing required before releasing tools for public use. In response, we advise strongly against exploiting this opportunity. While a language subset may be sufficient to compile small test programs, ultimately the run-time environment also requires library and system services whose code typically exploits most features of C. In particular, if an implementer wishes to reuse existing library or operating system code, an extensive implementation of C is needed with support for common non-standard extensions. This is the perspective we took in our work.

### 6.2.2 Language primitives vs. APIs

Once new features are introduced in a machine interface, two ways exist to exploit them from programs: encapsulation in APIs and embedding into the language via new primitive constructs recognized by compilers and associated new translation rules to machine code\(^1\).

\(^1\)In the case of C, extensions via the preprocessor’s #pragma feature is a form of embedding as it influences code generation.
Encapsulation is technically trivial, and it is desirable when porting existing software using established APIs such as the POSIX thread interface. However the following must be considered with the proposed architecture. As explained in chapter 4, a thread that issues a long-latency asynchronous operation, e.g. memory load or thread creation, uses regular ISA register names for the endpoints of the communication channels with the asynchronous operation. Meanwhile, the proposed interface creates the opportunity to interleave the asynchronous thread management operations (e.g. “allocate,” “create”) or inter-thread communication operations with other instructions from the same thread. To exploit this opportunity with encapsulation, the two phases must be part of separate API functions, and code generators must be configured to avoid reusing these ISA register names for computations while a thread management or communication operation is ongoing. Otherwise, any register spills between phases will cause the thread to wait prematurely for completion of the operation and waste an opportunity for overlapping instruction execution with the asynchronous operation. The same applies for synchronizers that hold the future of asynchronous completions (section 4.2): if the synchronizer that holds a future is spilled, this would cause the creating thread to wait prematurely on the asynchronous operation. To address these issues, a code generator would need to perform an inter-procedural register allocation; furthermore, if the API implementation is compiled separately from the application code, register allocation must then be deferred until all objects are available. Given that no publicly available compiler framework had support for link-time inter-procedural register allocation prior to our work, encapsulation seemed impractical with the new architecture and embedding remained as the unavoidable strategy.

That said, there is also a quantitative reason as to why embedding is more desirable. The architecture allows fine-grained, short-latency thread management and inter-thread communication; and the cost of diverting the control flow for a procedure call is large compared to the synchronization latency (e.g. 40 processor cycles to transfer control to a different procedure vs. 6 cycles to create a family of one thread and 0-1 cycle to communicate a scalar value from one thread to another). In this circumstance, the choice to embed the TMU primitives as new language primitives reduces the overhead to exploit the synchronization and scheduling granularity offered by the architecture.

6.2.3 General design directions

One design motivation that supported our work was to entice code generators and programmers to expose the fine-grained concurrency of numerical computations, even the partial concurrency available in dependent computations, in order to enable the automated mapping in hardware of all program fragments, even a few instruction long, to separate cores or hardware threads. This requires, for example, simplifying the replacement of sequential loops in programs by a construct that a) isolates simpler, non-repetitive communicating sub-sequences, b) instructs the underlying hardware to run the fragments on separate, simple processors/threads and c) instructs the underlying hardware platform to connect the processing units in a network of dataflow channels that corresponds to the computation structure. When this design objective is reached, it becomes possible to transform dependent loops in sequential programs by a network of dependent threads interleaved in the pipeline, removing the need for branch prediction to maximize utilization.

Another motivation was to promote resource-agnosticism, that is promote the expression of programs in a style where the semantics stay unchanged should the hardware parameters evolve. In particular, the approach should discourage programmers from assuming,
or knowing, or restricting at run-time the specific amount of effective parallelism (e.g. the number of processors or thread contexts available) when constructing algorithms. This is because otherwise the program is tailored to a specific hardware topology and must be re-designed upon future increases of parallelism. This requires language mechanisms that can express concurrency mostly via data dependencies and declarative concurrency, in disfavor of explicit control of individual thread creation and placement, and explicit inter-thread communication. When these features are used, it becomes possible to scale the run-time performance of a program by changing the amount of parallelism, and without changing the machine-level encoding of the program. Conversely, it also becomes possible to run any concurrent program on a single processor, since the program cannot assume a minimal amount of parallelism. Thus the flexibility required for dynamically heterogeneous systems (cf. chapter 2) is achieved.

We detail this second objective further in [PJ10]; they are shared with other language designs, such as Cilk [BJK+95] or more recently Chapel [CCZ07].

6.2.4 Mandating a sequential schedule

We considered the hypothetical situation where there were no theoretical or research challenges to the implementation of code generators and operating software services. We then considered what were instead the likely practical obstacles to evaluation of the platform. The motivation was to determine early on what the likely obstacles were, then integrate their avoidance as technical requirements.

We found that the validation of the infrastructure, which is the step immediately prior to evaluation, was the largest obstacle. Validation ensures that the various tools function as expected; that is, for each tool we can check for:

- completeness: valid input is accepted successfully;
- correctness: valid output is produced consistently from valid inputs;
- consistent rejects: invalid inputs are rejected consistently with informative messages.

By negating any of these aspects, we can enumerate all the possible failure modes which could be observed in a software stack with higher-level compilers targeting a C interface, in turn targeting the hardware platform:

incompleteness errors:

- \( F_1 \): the higher-level compilers fail clearly on valid benchmark program;
- \( F_2 \): the code generator fails clearly on valid input source code;
- \( F_3 \): the assembler / linker fails clearly on valid input assembly;
- \( F_4 \): the reference hardware implementation fails clearly on valid machine code;

incorrectness errors:

- \( F_5 \): the higher-level compilers accept valid benchmark programs and produce invalid code in the interface language;
- \( F_6 \): the code generator accepts valid input source code and produces invalid assembly code silently;
- \( F_7 \): the assembler / linker accepts valid input assembly and produces invalid machine code silently;
- \( F_8 \): the reference hardware implementation accepts a valid machine code and produces incorrect outcomes;
inconsistency errors ("false positives"):  
- F9: the higher-level compilers do not reject invalid benchmark programs and produce apparently valid code with invalid behavior in the interface language;  
- F10: the code generator does not reject invalid input source code and produces apparently valid assembly with invalid behavior;  
- F11: the assembler / linker do not reject invalid input assembly and produces apparently valid machine code with invalid behavior;  
- F12: the reference hardware implementation does not reject invalid machine code and produces unclear outcomes;  

compound failures: any combination of the above.

Based on personal prior experience with complex infrastructures, we established the need for clear methods to isolate failure modes and the components of compound failures as a primary requirement for any technical realization. In particular, we put the following goals at the highest priority:

- we needed to be able to clearly separate a) compounds of incorrectness errors upstream and consistent rejects downstream from b) incompleteness errors. This is crucial to establish responsibility upon failure. For example, if an error is observed at the code generator, the cause of the error can be either failure mode F5 or failure mode F2. In the former case, the responsibility lies with the provider of higher-level compilers, in the latter case with the interface language or below.  
- we needed to be able to clearly separate a) incorrectness errors at one level from b) compounds of incorrectness errors upstream with a stack of inconsistency errors downstream. This is crucial to estimate the cost to fix the error. For example, if an error is observed at the hardware implementation, the cause can be either failure mode F8 or a combination of failure modes F5 and F10 to F12. The pitfall is that peer operating software providers may be tempted to believe failure mode F8 and push the responsibility for the error entirely towards the platform, whereas work at all levels is actually necessary to resolve the situation.

Retrospectively, these concerns were not completely new. At the level of the hardware implementation, both the partners in charge of UTLEON3 and the author of MGSim had set up an environment where any machine code could be repeatedly run over multiple points in the hardware design space (e.g. varying number of cores, various memory architectures). This could distinguish invalid machine codes, i.e. "software errors" which would fail consistently across all design points, from incorrectness or incompleteness errors in the hardware implementation, i.e. "hardware errors" which would fail inconsistently across hardware design points. More generally, to support more detailed failure mode detection upstream across software layers, extensive unit test suites and regression tests are needed.

Beyond unit testing, we decided to place an extra requirement on any C language extensions: ensure that the C compiler can generate valid sequential code for any new language constructs towards existing (legacy) downstream tools and architectures, e.g. commodity desktop computers. This enables proper troubleshooting and analysis of behavior on existing computers. This requirement is crucial to validate the correctness of the interface-level test suite itself, and also to robustly distinguish inconsistency or incorrectness errors upstream, under the responsibility of the higher level compiler providers, from any incorrectness or incompleteness errors downstream, under the responsibility of the platform. This is akin to requiring Cilk’s faithfulness [Lei09] or Chapel’s serializability [Cra11].
6.3 Proposed code generator

After an initial step where we crafted various unit test suites as per the requirement above, we attempted to realize the semantics of the desired language constructs using an unmodified code generator and inline assembly: defining a thread program, defining dataflow channel endpoints, using family creation instructions from the ISA.

Using a serendipitously powerful combination of existing language features from the GNU C Compiler (GNU CC), we successfully implemented a freestanding C translation environment [II11b, 4§6], with extra language constructs to access the new hardware concurrency management features. We detail this experimental process and the corresponding code generation strategy in Appendix H.

To summarize, our implementation is based on a front-end with three phases. The first phase occurs between C pre-processing and translation: new syntax forms that look like function calls, i.e. keyword followed by parameters between parentheses, are substituted using context-free rules in M4 [KR77]. The substituted C code uses GNU CC extensions, including inline assembly. Then GNU CC’s legacy code generator towards the substrate ISA, i.e. without extensions, is invoked with flags to instrument the register allocation and control the visible synchronizer windows (cf. sections 3.3.3 and 4.3.3). Finally, a post-processor modifies the assembly code generated by GNU CC to adhere to the ISA semantics, and extends it with control bit annotations for thread switching (section 4.4).

The new syntax forms thus form new language primitives that extend the C substrate language. These language primitives expose fewer semantics than allowed to programs by the machine interface (chapter 4); they also intentionally erase the distinctions between the various hardware implementation variants, so as to be translatable to all of them from a single source representation.

The resulting compiler technology is able to input a program source expressed once using this interface language, and emit code for various implementations of the new architecture and for a legacy sequential processor, as required in section 6.2.4. We call this interface language “SL,” which we describe in detail Appendix I, and whose main constructs and their translation are illustrated in table 6.1 and fig. 6.1. We further ensured that the main command-line tool that controls the transformation pipeline, called slc, has the same interface as an existing compiler command (gcc); this way, we were later able to successfully reuse it as a drop-in replacement in existing build systems for legacy software, for example existing C library code needed to run benchmarks as described below in section 6.4.
Construct | Description
--- | ---
`sl_def(name, channels...) { body... } sl_enddef` | Define a thread program. The signature declares a static set of channel endpoints.

`sl_create(range, name, channels...)` | Perform bulk creation and synchronization on a named thread program with “source” values for the dataflow channels. Both parts form a single syntactic construct.

`sl_glparm(type, name), sl_glarg(type, name [, v])` | Declare a “global” dataflow channel endpoint. Types are manifest. The “a” version optionally provisions a source value.

`sl_shp parm(type, name), sl_sharg(type, name [, v])` | Declare a pair of “shared” dataflow channel endpoints. Types are manifest. The “a” variant optionally provisions a source value for the “leftmost” channel.

`sl_getp(name), sl_geta(name)` | Read from the named endpoint of a dataflow channel.

`sl_setp(name, v), sl_seta(name, v)` | Write to the named endpoint of a dataflow channel.

Table 6.1: Main constructs of the resulting SL language.

A specification is given in Appendix I; example uses are given in Appendices J and K;

![Diagram of SL tool driver](image)

Figure 6.2: Overview of the placeholders in the SL tool driver.

### 6.3.1 Compiler combinators

A remarkable aspect of the SL tool chain, one originally motivated by short-term practical requirements, is that it does not modify the underlying C compiler used to generate the assembly code from the input source. Instead of going the “traditional route” of language extensions by modifying an existing compiler and introducing new features throughout the front-end, middle-end and back-end, we instead subverted the compilation pipeline of the existing C compiler to “trick” it into creating valid code for the new architecture.

A bird’s eye overview is given in fig. 6.2: we first let the underlying compiler pre-process the code, then we perform source-to-source transformations on the pre-processed code to insert the new machine constructs via inline assembly and other techniques, then we let the underlying compiler process the code, and finally we post-process the generated assembly source to ensure its validity to the target machine interface. We detail this scheme in Appendix H. We then made it simple to configure which underlying compiler is used and the set of rules for the upstream code transformation and downstream assembly post-processing.
We would like to suggest that the resulting framework is a compiler combinator: given an existing C compiler, we can combine the existing compiler with a set of rules and our tool chain to obtain a new compiler for a different target and/or an extended input language. With this methodology, we were later able to quickly support three different implementations of the new architecture (a 64-bit platform with the Alpha ISA and two 32-bit platforms using the SPARC ISA) and sequential execution by simply changing the operands of the combinator, as illustrated in fig. 6.3.

We consider the generality of this approach, together with our ability to reuse existing code generators without changes, to be a confirmation that the new architecture’s design can be considered as an add-on improvement on existing ISAs, instead of a radical new approach to programming parallel processors at a low level. As we show in the remainder of our dissertation, the originality of the proposed design is found at a higher level, with memory semantics and resource management. By choosing an implementation strategy that keeps away from complex approaches to code generation, we were able to invest more effort into addressing system-level issues. Besides, our strategy does not sacrifice performance: the benchmark results in chapter 13 show that our non-intrusive, blunt approach to code generation was sufficient to exploit the architecture’s fine-grained concurrency management features.

The generality of the framework was confirmed again later with the successful re-targeting of the SL tool chain to other software-based concurrency management frameworks [vTJLP09, Mat10, UvTJ11] for existing hardware architectures.

6.3.2 Discussion on the generality of the approach

As we detail in Appendix H, to achieve our implementation we used standard features from [II99, II11b] and a tool box formed of carefully selected pre-existing language extensions from the GNU C Compiler (GNU CC):
• type inference with the `typeof` keyword [Fred], to produce declarations to new variables with the same type as an existing variable, for example:

```c
__typeof__(a) b;
```

• inline assembly with the `asm` keyword with access to enclosing local C variable from the assembly code [S.03, Frea], for example:

```c
__asm__("mov %1,%0" : "=r"(b) : "r"(a));
```

• explicit machine registers for local variable declarations [Free], for example:

```c
int a __asm__("$1");
```

• the ability to exclude selected register names from register allocation [Frec], for example using the command-line argument `-ffixed-$1`.

We investigated whether our implementation was tied to GNU CC due to the use of these non-standard language extensions, or whether the strategy could be adapted to other compiler technologies. The purpose of this investigation was twofold. First, we acknowledge a recent trend (post-2010) that promotes LLVM as a substitute to GNU CC for general-purpose programming, and we wish to ensure a future transition path, should one become necessary. Second, we wish to highlight that the successful exploitation of the architectural features was not dependent on this specific compiler technology, so as to enable third parties to develop their own compiler technology instead.

We found active support for `typeof` also in IBM’s and Intel’s compilers [IBMb, Intb], and in ARM’s compiler until version 3.0. Although this construct for type reuse in C variants is still relatively uncommon, we expect support for this feature to become prevalent once the new C++ language standard [II11a] is adopted by vendors: most industry-grade frameworks share features between their C and C++ compilers and the new standard mandates support for the C++ `decltype` keyword with identical semantics as `typeof`.

Support for the `asm` keyword has been historically pervasive across implementations of C. It is the main means provided to C programmers by the compiler implementer to bypass the compiler and emit machine instructions explicitly. We found active support for both `asm` and the use of C variables in inline assembly in particular in Microsoft’s Visual C/C++ compiler [Cor], Oracle’s (previously Sun’s) Solaris Studio compiler [Ora11], IBM’s XL compiler [IBMa], Intel’s C/C++ compiler [Intc], ARM’s compiler [ARMa], LLVM and its Clang front-end [LLV], and Open Watcom [Con08].

Support for explicit mapping of variables to registers is less prevalent but not unique to the GNU implementation either. We found that it is supported by Intel’s and ARM’s compilers using the same syntax as GNU [Intb, ARMb] and in Open Watcom via pragmas.

Finally, we also found support for customizable register sets during register allocation in Intel’s compiler (`-mfixed-range`, [Inta]). The set of available machine registers is also expressed explicitly in the source code of Open Watcom’s and LLVM’s back-end code generators, so one could manually modify this set and generate multiple compilers with different register uses in each. We expect that this technical modification could be realized in proprietary compilers as well, if need arises.

To summarize, although we were fortunate to find this conjunction of features and their flexibility in the GNU implementation, we can reasonably expect that other compilers support similar features, or can be modified to support them at a low cost.

### 6.3.3 The need for manifest typing

The reason why we separate the syntax to declare integers and pointer thread arguments (`sl_glparm`, `sl_glarg`) from the syntax for floating-point (`sl_glfparm`, `sl_glfarg`) is that
they must be translated to different register classes in the inline assembly, and our translation
strategy cannot analyze C types.

This syntax is undesirable because it exposes a feature of the underlying ISA (namely,
different register classes on Alpha) that C was originally intended, by design, to hide. Moreover,
this proposal is incomplete as support for C’s long double and _Complex types require
other register classes or register pairings on most ISAs which would require yet additional
SL syntax with the current implementation.

Instead, to alleviate this issue and properly hide register classes without introducing
type analysis in the source-to-source transformer, we suggest that future work use the new
_Generic construct from [III1b, 6.5.1.1] which can select different expressions based on the
actual type of a conditional expression. This mechanism would enable delegating the task
of choosing the right implementation based on the type information to the underlying code
generator.

6.3.4 Why “create” and “sync” are bound

A characteristic SL feature, detailed in Appendix I.5.8.1, is the binding between the words
“sl_create” and “sl_sync” in one syntax rule for C’s block item. This implies, for example,
that the forms in figs. 6.4a and 6.4b are invalid, whereas the form in fig. 6.4c is valid.

There are three motivations for this, one fundamental and two practical.

The fundamental motivation is to hide the semantic distinction between “fused creation”
ISAs and “detached creation” ISAs introduced in section 4.3.1.2. To hide this, we specify
that the created work starts “no earlier than sl_create” and “no later than sl_sync” in the
control flow. This is to provide the understanding to the programmer that either part of
the construct is ultimately responsible for the actual creation, without specifying which. If
the language had allowed to omit sl_sync, as in fig. 6.4a, we would not be able to generate
code for fused creation ISAs where the information needed for creation, generated by uses
of sl_seta, is only fully known at the point sl_sync is reached. We detail this further in
Appendix G.3.

The first practical reason is that each use of sl_create expands to both multiple state-
ments at the point where it is used, and new variable definitions at the beginning of the
current scope. The scope is identified by the last preceding occurrence of the left brace “{” in
the program source. The new variables are used both by the expansion of sl_create
and subsequently by the expansion of sl_sync. If the language would allow sl_sync to
occur in a higher-level scope than sl_create, then the variable declarations would need to
be pulled upwards to the first common scope. This is not possible in our setting, because
some of the data types involved may be defined with typedef in the inner scope. A proper
treatment of this would require full type analysis of the program source, i.e. a full-fledged
C front-end, which is what we tried to avoid in the first place.
The other practical reason is that if the language would allow the pair “sl_create-s1_sync” to occur in a position in the C syntax which only accepts statements (as in fig. 6.4b), we would confuse text editors and other source analysis tools which assume that the semicolon is a statement separator.

6.3.5 Opportunities for platforms with detached creation

Setting aside the aforementioned requirement to support fused creation ISAs, and focusing instead exclusively on ISAs featuring detached creation, an opportunity exists to expose this flexibility in the language interface.

We explored this opportunity as follows. First we extended the pair “sl_create-s1_sync” with a new form “sl_create-s1_detach” with the same lexical structure as described above in section 6.3.4, but where the bulk synchronizer is released upon sl_detach without synchronization (fig. 6.5a).

We also defined a new statement sl_spawn which performs bulk creation, sends source values for the dataflow channels, and produces the identifier for the remote bulk synchronizer into a variable. We then defined a new statement sl_spawnsync that combines both synchronization on the named bulk synchronizer and releasing the bulk synchronizer. We summarize this in fig. 6.5b. As is, this construct does not allow spawned workloads to use the “shared” dataflow channels from section 4.3.3.3, because the argument list of sl_spawn, necessary to declare the temporary variables to hold the channel sink values, may not be visible from sl_spawnsync.

Through discussions with our users, we obtained the following feedback:

- the behavior of sl_spawnsync is only properly defined if it is used only once by a program on a given bulk synchronizer. This is because after a first occurrence releases the synchronizer, any further occurrence would use a stale, invalid synchronizer identifier. The hardware protocol is not designed to handle this situation. Because of this, the definition of proper functional futures [Hal85] using sl_spawn...sl_spawnsync with multiple uses of the second half would require an additional flag to test the validity of the bulk synchronizer, which must be updated atomically throughout the system at the first run-time use of sl_spawnsync.
- a problem exists if the argument list of “sl_create...sl_detach” or “sl_spawn” does not fit entirely through hardware synchronizers. Indeed, our implementation escapes arguments to the call stack, as we will describe in chapter 8 for sl_create...sl_sync; this approach can only be used if the lifetime of the activation record of the creating
thread extends beyond the lifetime of the created thread(s). This is only true if \texttt{sl_spawnsync} is executed before the escaped arguments are cleaned up, or the created workload terminates before the creating thread. To solve this the language feature must be constrained in either of the following ways:

- \texttt{sl_spawnsync} must appear within the same C scope at \texttt{sl_spawn}; or
- \texttt{sl_spawn} and \texttt{sl_create...sl_detach} are restricted to only accept \( N \) arguments, where \( N \) is an implementation-specific constant that describes the number of channels physically available; or
- \texttt{sl_spawn} and \texttt{sl_create...sl_detach} allocate extra arguments on a shared heap. Then either the first use of \texttt{sl_spawnsync} (for \texttt{sl_spawn}), or the termination of the created work, whichever happens last, becomes responsible for the deallocation of the argument data from the shared heap.

The spectrum of solutions to these issues involve complex trade-offs between usability and performance which we did not explore fully. Lacking understanding of these trade-offs, we chose instead to not advertise these constructs as an official language feature. Instead, we used them as a system-oriented feature (i.e. hidden from applications) to implement some of the operating software services described below in section 6.4.

6.3.6 Discussion on the language design and future work

We do not claim any innovation in terms of programming language design. We acknowledge that the proposed language constructs are crude and tailored to the specific architecture considered. Their semantics are a mash-up of concepts borrowed from traditional fork-join parallelism, bulk-synchronous parallelism, Cilk, and previous work from our research group, selectively chosen to facilitate the demonstration of the proposed machine interface.

A scientific contribution should be found instead in the careful choice of syntax which enables straightforward code generation from the same source semantics to a diversity of platforms, using a relatively simple wrapper around existing compilers (section 6.3.1).

Since the advertised innovation is to be found in the machine interface, it follows that a diversity of existing programming languages could be extended or ported towards the proposed architecture. For example:

- the parallel loop constructs with static scheduling from OpenMP [Ope08], independent loops in Fortran, and the parallel loops from Intel’s TBB [Rei07], map transparently to bulk creation and synchronization in our platform. The variants of these constructs that require dynamic scheduling can be translated to a program using the run-time scheduler from [SM11].
- the fork-join constructs from Cilk can be mapped onto \texttt{sl_spawn/sl_spawnsync} such as described in section 6.3.5. The issues we identified with \texttt{sl_spawn/-sl_spawnsync} would not apply because Cilk forces its “sync” construct to be in the scope of its “spawn” constructs and does not allow multiple “sync” occurrences for each “spawn.”
- the vector operations from OpenCL [Khr09] can be translated to bulk creation and termination of threads running programs that implement the corresponding OpenCL operators. OpenCL’s explicit memory types can be discarded as there is only one shared memory system.

Note that the existing languages’ semantics cannot be translated to SL source text, in other words SL cannot serve as a common intermediate representation for the diversity of
concurrency constructs in existing languages. This is because these languages also contain additional data types, synchronization devices and scheduling hints which do not have equivalents in SL, although they could be implemented by directly targeting the machine interface from chapter 4 in a dedicated code generator.

To conclude, we do not advertise our SL extensions as the only possible interface language to the target architecture. Common interfaces, if any were defined, would likely exist as the internal intermediate representations in compilers’ middle-ends. Instead, we propose our work as a modest yet practical vehicle to bootstrap further research in this environment.

### 6.4 Operating software services

We subsequently used the aforedescribed new compiler technology to port operating software code as described in the sections below.

#### 6.4.1 Minimal run-time requirements

The minimal run-time environment required by the test and benchmark applications we considered includes support for:

- the following components from the standard C library: math functions, string manipulation, console output (`stdout, stderr`), heap allocation.
- the following POSIX services: abnormal asynchronous termination (`abort`), file I/O (`read/write`).

Autonomous services from the C library can be obtained cheaply by reusing standard F/OSS implementations, as we describe below in section 6.4.3. Existing heap allocators mostly require only the traditional service `sbrk` which can be implemented cheaply on a flat address space. In contrast, console and file access are a more costly requirement because they imply full-fledged support for external I/O channels, persistent storage and filesystems. To implement these cheaply, we used the heterogeneous integration with companion cores described in section 5.5.1.

#### 6.4.2 Operating system services

- We used companion processors to support file access: we implemented “proxy” functions for the base POSIX calls `open, close, read, write, link, unlink, sync, dup, dup2, getdtablesize, fsync, rename, mkdir, rmdir, stat, fstat, lstat, opendir, fdopendir, rewinddir, telldir, seekdir, closedir, readdir`. This was sufficient to implement the streams and file I/O APIs from the standard C library and support direct uses of the system interface for file access by programs. We also wrapped the supplementary X/Open “parallel I/O” operations `pread` and `pwrite`, because contrary to `read / write` these provide the file position explicitly and can thus be issued concurrently from separate threads without interference. We surmise that extended file access APIs such as POSIX’s asynchronous I/O interfaces (`aio_read, aio_write`) can be added similarly at minimum cost.

- Several system services, in contrast, did not require to delegate the behavior to the companion processor: system memory allocation (`sbrk` and anonymous mappings with `mmap`) can run locally on the microthreaded cores to support C’s `malloc`. Time and date functions (`gettimeofday`, which support C’s `time`) can access a RTC device directly. We also used the substrate ISA’s time-stamp counters to support C’s `clock`.
To implement console input-output (POSIX’s special file descriptors 0, 1, 2), we implemented a configuration option which selects between various I/O device combinations (e.g. either the UART or matrix display devices).

The following features were designed but are not yet implemented:

- for dynamic process creation (\texttt{vfork+exec}) and process data encapsulation, a loader can make a copy of the initial program data segments in the shared address space, a strategy suitable with an Alpha ISA substrate, which uses GP addressing, or alternatively the entire program image is duplicated and re-linked, as suggested with the default image format for the SPARC ISA substrate;
- the environment variables (for C’s \texttt{getenv}) are loaded from ROM for the first program, and subsequently inherited upon further process creations.
- during program initialization, the loader creates a new thread of execution from the program’s entry point, which maps to the C library initialization routine, which in turn initializes the C library’s internal state (e.g. \texttt{malloc}’s allocation pool), runs any program-defined data constructors and finally transfers control to the program’s \texttt{main} function.

Instead, in our implementation we provided a simple loader able to execute a single monolithic program with both application and operating software in a single memory image. We suggest the features above as future work.

6.4.3 C library services

To avoid re-implementing a C library from scratch, we studied how to reuse existing code. To select a code base, we pre-selected existing implementations satisfying the following requirements:

- \texttt{available in source form}, either in C or assembly source for one of the target ISAs, because it had to be recompiled/re-assembled to the binary format suitable for the new architecture;
- \texttt{suitable for cross-compilation}, since the platform where the code was compiled would not be the target architecture;
- \texttt{suitable for execution on a shared memory heavily multithreaded platform}, since this is the basic setting of all selected benchmarks;
- \texttt{modular}, since not all library functions would be needed and some could not be possibly supported initially anyways, such as support for file access;
- \texttt{compiler-agnostic}, since we could not guarantee support for specific existing non-standard C extensions beforehand.

We then analyzed those few codebases that satisfy requirements \texttt{L1} and \texttt{L2}: newlib [Red], \texttt{µClibc} [YMBYG08, pp. 115–127] [And], the GNU C library [Freb], the standard C library of BSD systems [MBKQ96, McK99], the standard C library of OpenIndiana\footnote{OpenIndiana was forked from OpenSolaris when Oracle discontinued the OpenSolaris project.} [JIS].

Our results are summarized in table 6.2. The implementations from GNU and OpenIndiana needed to be excluded early because of requirement \texttt{L5}. We then considered the following choice: either start from a thin embedded implementation and improve it to ensure that it would be suitable for use in a large multithreaded setting, or start from a larger
CHAPTER 6. PROGRAMMING ENVIRONMENT

Table 6.2: Requirements satisfied by existing C library implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
</tr>
</thead>
<tbody>
<tr>
<td>newlib</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>µClibc</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>BSD</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>GNU</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>OpenIndiana</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

The listed implementations all satisfy requirements L1 and L2.

implementation that satisfied requirement L4 and reduce its dependencies on underlying operating system support.

We adopted the latter approach, starting from the BSD components, for the following reasons:

- analyzing the uclibc and newlib implementations revealed multiple shared global-scope variables and a design made with the assumption of sequential execution; adding the necessary state protection to arbitrate state access during concurrent execution seemed an arduous task, with the associated high risk of errors that would add a potentially costly troubleshooting overhead to the benchmarking efforts;
- due to its longer history and wide audience, the BSD library is more comprehensive feature-wise and thus its selection reduced a priori the risk that our choice would fail to support any additional application requirement discovered in a later phase of our research.

By reusing components from the FreeBSD project [MNN04] with few changes, we were able to provide comprehensive support for the following C standard services: diagnostics (assert.h, [II99, B.1]/[II11b, B.1]); character handling (ctype.h, [II99, B.3]/[II11b, B.3]); errors (errno.h, [II99, B.4]); characteristics of floating types (float.h, [II99, B.6]); sizes of integer types (limits.h, [II99, B.9]/[II11b, B.9]); mathematics (math.h, [II99, B.11]/[II11b, B.11], support for long double omitted); variable arguments (stdarg.h, [II99, B.14]/[II11b, B.15]); boolean types and values (stdbool.h, [II99, B.15]/[II11b, B.17]); common definitions (stddef.h, [II99, B.16]); integer types (stdint.h, [II99, B.17]); buffered output functions, including formatted output, on standard output streams and character strings (stdio.h, [II99, B.18]/[II11b, B.20]); a subset of the general utilities (stdlib.h, [II99, B.19]/[II11b, B.21]); a subset of the string handling functions (string.h, [II99, B.20]/[II11b, B.23]); a subset of the date and time functions (time.h, [II99, B.22]/[II11b, B.26]).

The remaining standard library services from [II99] and other newer features from [II11b] were not implemented because they were not needed by the test applications we considered. If they were ever needed, the following could be imported from existing codebases at little cost: extensions from [II11b] to the services already listed above, format conversions of integer types [II99, B.7]/[II11b, B.7], alternative spellings for operators [II99, B.8]/[II11b, B.8], localization [II99, B.10]/[II11b, B.10], alignment operators [II11b, B.14], standard input and file access [II99, B.18]/[II11b, B.20], the arithmetic general utilities [II99, B.19, abs, div, etc.]/[II11b, B.21], multi-byte/wide character utilities [II99, B.19,B.23,B.24]/[II11b, B.21,B.27,B.28,B.29], and type-generic math functions [II99, B.21]/[II11b, B.24], the “no return” function specifier from [II11b, B.22].

In contrast, the following services would require more research and effort:
• support for complex arithmetic (`complex.h`, [II99, B.2]/[II11b, B.2]) and long floating types (`long double`) would require extra attention during code generation as they require more than one machine register (or synchronizer in the proposed architecture) for a single variable;
• support for controlling the floating point environment (`fenv.h`, [II99, B.5]/[II11b, B.5]) requires a hardware interface to control the FPU in the platform;
• non-local jumps (`setjmp.h`, [II99, B.12]/[II11b, B.2]) require platform-specific, hand-coded assembly code as they cannot be expressed directly in C;
• signal handling (`signal.h`, [II99, B.13]/[II11b, B.13]) conceptually conflicts with the purpose of hardware microthreading, a topic which we revisit later in sections 14.2 and 14.5;
• the process management interfaces (`system`, `atexit`, [II99, B.19]/[II11b, B.21]) require operating system support for separated processes, a topic which we revisit later in sections 14.4 and 14.5;
• support for atomic object access and threads introduced in [II11b, 5.1.2.4, B.16, B.25] does not map directly to the proposed concurrency management features, a topic which we revisit in chapter 7.

To summarize, the large compatibility of the proposed C compiler with legacy C code allowed us to reuse large existing code bases for operating software components. The resulting combination of compiler and library code provides a large subset of the standard hosted C execution environment from [II11b, 4§6, 5.1.2.2].

Summary

We have acknowledged prior work which has attempted to provide an interface language to the proposed architecture, and we demonstrated the shortcomings of this approach. We have outlined the design requirements for a machine interface language derived from C. We then adopted a practical implementation strategy based on maximum reuse of existing tools. Using this strategy, we succeeded in providing a C compiler for the target architecture. We also provided a meta-compiler that implements transformations of a C language extension, which we call “SL,” towards multiple platforms including the proposed architecture from part I. While our implementation exploits compiler features specific to GNU, we also show that the strategy is portable to other compiler substrates. Then using our freestanding programming environment, we subsequently ported existing operating software components suitable to run hosted C programs on the platform introduced in chapter 5. We also outlined how to extend this software support in future work.