On the compilation of a parallel language targeting the self-adaptive virtual processor
Bernard, T.A.M.

Citation for published version (APA):
Enschede: Print partners Ipskamp

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating
your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask
the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam,
The Netherlands. You will be contacted as soon as possible.
Chapter 6

Implementing the SVP compiler

Compilers are the Queen of Computing Science and Technology. They have long been the bridge from applications to systems, but now they determine which architectural features should be implemented in new hardware, as well as which new language features will be effective for software developers.

David J. Kuck, Intel

While Chapter 4 has discussed the basics of conventional compilation and also advanced SVP compilation, Chapter 5 has reported the hazards on compiler optimizations with SVP properties. Methods and solutions to expose the SVP properties in imperative-language compilation generated considerable research effort that this chapter presents as an experience report.

In this chapter, we discuss effort invested in designing and extending our experimental compiler. The role of the compiler in the SVP system is critical, because it is the bottleneck within the SVP parallel computing system bridging two worlds: the software and the hardware. The pressure on the compiler

The contents of this chapter are based on these publications:

is therefore very high. In that context, we discuss design decisions and their impact in compiler research including limitations due to the experimental environment. One relevant question, for instance, concerns reasons not to write a new compiler from scratch instead of extending an existing framework. Another question concerns the selection of the research compiler platform. We then discuss these compilation issues related to generic issues with concurrency and specific issues with the Microgrid platform.

6.1 Role of the compiler

As part of any computing system, the role of the compiler is often misunderstood in terms of scopes of its properties and functionality. What a compiler does and is supposed to do is straightforward to developers: tell whether my code is correct, if so, improve it for this platform, but also, just to make it executable. Nevertheless, modern imperative-language compilers are, in general, very large pieces of software systems and are seen as black-boxes by software developers as discussed in Section 4.3. Their advanced infrastructure enables maximal modularity in combinations with language front-ends and target platform back-ends. In addition to that, to please developers, compilers contain a huge variety of passes comprising transformations and optimizations to take maximal advantage of any piece of input programs. Consequently, compilers are usually evaluated with the following criteria:

- **speed**: for large software system to compile such as the compiler itself, the speed of compilation is relevant to developers to permit a convenient time-frame while software development.

- **space**: while compiling a program, compilers deal with huge amounts of data and memory structures which directly impact on the previous criterion about compilation speed. Therefore, using an optimal amount of memory space while compiling is a major criterion for an efficient compiler.

- **feedback/diagnostics**: developers are interested in a compiler that can let them know whether their program is correct. Therefore, the quality of diagnostics is another relevant criterion for selecting the appropriate compiler for any software development.

- **debugging**: based on diagnostics and feedback, compilers often offer inter-stage dumping representation involving extra information for advanced users which can pinpoint problem locations in the code.

- **compile-time code efficiency**: this criterion is based on a combination of compiler components such as register allocation, instruction scheduler, etc. The quality of code directly depends on back-end features and optimization performance. The code size of an input program can be completely
different, depending on whether the instruction scheduler is efficient for a specific architecture.

### 6.2 Compiler design decisions

We had to make design decisions to get results as soon as possible: considering research time, implementation, experiments, etc. This section summarizes them and discusses the impact on the research. To give some perspective on the complexity of compiler research, the source-to-source SAC compiler has been created from scratch by a crew of scientists in the 1990s. The idea was to generate a compiler for a functional language targeting the C language. Requirements for a compiler were therefore different for this project where the target is static and the language only evolves along with compiler’s features. Ever since, the SAC compiler gained maturity and new features with time and considerable effort. In the context of this thesis, the language and architecture features evolved during compiler development undertaken by a reduced crew (one PhD student as permanent researcher and developer and temporary support from other CSA group members). Because of time constraints and available manpower, we decided not to start a new compiler from scratch. Instead, we looked for a decent candidate as compiler framework to be extended and used as proof-of-concept. Writing a compiler from scratch is simply out of scope for this research. Moreover, there are no relevant scientific interests to reimplement existing technology that would consume a considerable amount of time.

#### 6.2.1 Compiler selection

We have selected an imperative-language compiler for our research after investigating a set of available compilers from industry and academia. Several criteria are relevant to allow a rapid deployment and implementation, then focus on the important topics of his research. Besides selecting a compiler framework, time constraints of this research and the available manpower influenced the choice to have an active community of compiler developers and researchers. To reduce the overhead of reimplementation, there are also technical constraints related to the presence of a C front-end and an Alpha back-end. A big advantage would be having a modular and easy-to-modify compiler such as a compiler framework. Open-source software systems are interesting where there are no copyright constraints and all the source code is available. In the long run, we aim to have a candidate that can support 32- and 64-bit instruction set to enable retargeting other instruction sets for further investigation.

---

1 Summarized from a private communication with Dr. C. Grelck
Chapter 6. Implementing the SVP compiler

The CoSy compiler

The CoSy compiler framework [79] is an advanced modular compiler framework. The system is based on a central internal representation (CCMIR) which is an interface to all internal compiler components, named engines. These engines analyze, read, write and rewrite to the CCMIR. The whole concept is to obtain a composable compiler depending on requirements of a specific language, architecture and functionality. With this unorthodox design, the CoSy compiler is far ahead of its competitors in terms of flexibility to target new architectures, insert new engines and deploy a production-quality compiler extremely fast with industrial standards. In theory, the CCMIR could be annotated to hold concurrency information and specific engines can be added or modified to support SVP properties. Nonetheless, in the context of this thesis’ research, there was a need for an Alpha back-end. The Microgrid target simulator already existed and employed an extended version of the Alpha instruction set. Unfortunately, developing an Alpha back-end from scratch was not relevant for the research topics of this thesis and the overhead was considered too high with restricted research time constraints (length of a PhD program).

The SUIF compiler

At first, the SUIF compiler [80, 81] was considered as a candidate for this research. We eventually turned it down because of a lack of documentation and support from an active community. When this research started (in early 2006), the compiler was not supported anymore. This compiler is designed to enable program analysis to parallelize sequential programs for shared-memory multiprocessors.

The Open64 compiler

Open64 [82] is an open-source research compiler targeting a 64-bit architecture. Open64 is a suite of optimizing compiler development tools designed for Intel Itanium Processors. It employs 5 IR layers which get closer to machine language the further the compilation process proceeds. This gives a distinct compartmentalization where new components can be added at different points of compilation. Open64 has a C/C++ and Fortran 90/95 front-ends. However, there is no Alpha back-end in this framework. University of Delaware (USA) is the gatekeeper of this project; nonetheless, no activity nor support were present when this research has started.

The ORC compiler

The ORC compiler [83] is a branch of the Open64 compiler. This compiler targets the IA-64 architecture specifically the Itanium Processor Family. This
project is based on the collaboration between Intel Corporation and the Chinese Academy of Sciences. The input languages are C/C++ and Fortran. The main purposes of ORC are new optimizations to better utilize Itanium architectural features and new features to facilitate future research. Similar issues to Open64 are present about lack of support and documentation, plus lack of an Alpha back-end. Therefore, we decided not to use this framework.

The LLVM compiler

Partly based on the GCC toolchain, the LLVM compiler became more popular in the compiler domain in the last few years. At the beginning of this research, this compiler was not mature enough to be considered as a candidate. The initial motivation was to be more aggressive and more efficient than the GCC compiler with a simpler infrastructure. Consequently, some parts of GCC have been reused such as different language front-ends. A convenient advantage with LLVM is the intermediate form (IF) that can be used to interface with another tool and be fed into the LLVM framework. This IF is an interface with external tools such as the GCC toolchain. If we had to choose again a new framework now rather than at the beginning of this research, LLVM would probably be the candidate we would be using.

The LCC compiler

Little C Compiler (LCC) [51] is a small retargetable compiler for personal use and educational purposes: simple to understand with documentation. There are sufficient back-ends, including an Alpha one, which makes this compiler a decent candidate. However, it is limited; there is no real intermediate representation nor advanced optimizations embedded in this software system. Therefore, there is little interest to use it for compilation investigation. The compiler candidate needs to have a longer-term development and support scheme.

The GCC compiler

The GNU Compiler Collection (GCC) [54] is one of the best known compilers in the market. Present on most UNIX-based systems, it is an open-source project targeting 32- and 64-bit architectures (back-ends: Alpha, MIPS, SPARC, x86, etc.). The input languages are C, C++, Objective-C, Fortran, Java, Ada, etc. Newer versions of GCC are frequently released and supported by a large community of developers and users. New features and corrections appear at each release. The good point for GCC is that it is very suitable for a wide range of platforms. On the other hand, it is very difficult to understand its infrastructure and to modify it easily and quickly for our project with a small development team. Up-to-date documentation is still a problem as with any very large software system [84]; the presence of an active community with a quick response
Chapter 6. Implementing the SVP compiler

time makes it a decent compromise. This imperative-language compiler is on
the scene for 20 years; it has evolved with an important set of optimizations and
efficient code generator which still make GCC an efficient compiler. A new reg-
ister allocator (integrated register allocator) \[85\] has indeed made big improve-
ments plus optimizations such as auto-vectorization. Nonetheless, GCC suffers
from its infrastructure design decisions; the compiler shows its age. Conse-
quently, extending it requires a lot of effort and time spent on verification and
validation. This contrast became clear when we observed the difference in flex-
ibility between CoSy and GCC while generating a dedicated compiler. CoSy
enables that with its modularity-based construction scheme, whereas GCC can
perform similar tasks but with more engineering effort. Nonetheless, recent
effort is being done to modernize the entire infrastructure with the ICI plug-
in initiative \[86\] and with the Modular-GCC initiative \[87\]. This compiler was
chosen for the reason that it was the least worst compiler from the ones evalu-
ated.

6.2.2 Impact of design decisions

Approach to SVP compiler conception

The conception of the compiler prototype follows the underlying SVP idiom to
expose explicit concurrency in the source code (i.e. $\mu$TC). Obviously, the front-
end must be able to parse and interpret the new constructs in the input lan-
guage. We can either follow a compiler improvement such as the work realized
in GUPC \[29\] where a minimum of compiler components have been extended
and a library has been added to the GCC framework. In that case, minimal
changes have been performed in the front-end to support the new keywords
of the UPC language. However, in the context of our research, there are needs
to extend deeper in its infrastructure in order to support SVP properties and
the Microgrid architecture: extension of the Alpha back-end and back-end gen-
erator with SVP assumptions. In theory, $\mu$TC constructs directly map onto in-
structions for the Microgrid platform. By the way, OpenMP is implemented in
a library within the GCC framework \[32\]. Consequently, at compile-time, the
compiler invokes this library each time it deals with concurrency constructs. In
our compiler implementation, we made the design decision to fully embed the
SVP concurrency idioms in the compiler’s internals. Although, this integration
requires more engineering work to get it to work, we believe in a long-term
vision to support concurrency natively in the compiler framework. As a result,
concurrency constructs are fully exposed to the compiler which can optimize
the program’s representation with extended optimizations. This decision has
partially caused the problems presented in this thesis. Conventional optimiza-
tions that work very efficiently can be reused to enhance program’s code with
respect to SVP semantics. Reusing compiler technology avoids wasting time
reimplementing it from scratch in the context of this research.
Chapter 6. Implementing the SVP compiler

GCC structure

Figure 6.1 illustrates the partition of GCC components over the entire framework. We have calculated the lines of code (LOC) for each file of the compiler with SLOCCount [88]. We use as platform GCC 4.1 Core Release which is a special release of the compiler toolchain with one single C front-end, all existing back-ends, and all optimization passes. The C front-end (FE) has a rather small size (around 4% of the overall amount) compared to GCC’s back-end generator and back-end descriptions (BE) (65% of the software system). GCC has more than 30 working back-ends including x86, IA64, Alpha, MIPS, etc. The infrastructure (INFRA) with a 11% size gathers garbage collection methods, data structure, debugging methods, compiler driver, etc. that are utilized in the entire framework. With this framework, we could retarget other back-ends in the future to evaluate the SVP architecture implementation with various instruction sets.

![Figure 6.1: Compiler composition of GCC 4.1 Core Release. Partition of code lines for each major GCC component: front-end (FE), middle-end (ME), back-end (BE), libraries (LIB), infrastructure (INFRA). Measures calculated on an unmodified GCC 4.1 core release with a single C front-end, standard back-ends, and standard optimizations. These measures are calculated with SLOCCount [88].](image)

With Figure 6.1 the GCC 4.1 framework shows a dominance of back-end code. The reason is mainly due to a large number of back-ends and a back-end generator that allows the addition of new back-ends. GCC 4.1 compartmentalizes optimization passes and transformation passes; this makes rather easy the addition of new optimizations. This compiler is clearly designed for adding new optimizations and new back-ends. Nevertheless, as soon as we step out of this paved road, the compiler internals are obscure with large and undocumented code files. Moreover, heavy inter-file dependencies are present which make the overall infrastructure troublesome.
Chapter 6. Implementing the SVP compiler

Analysis of SVP extension

Selecting GCC as our research compiler framework allows the reuse of advanced compiler technology. Nonetheless, the scientific question is to know whether this technology can be reused when integrating concurrency idioms in the compiler’s internals. We have calculated and summarized in Figure 6.2 the effort we have invested in each compiler component. Although measured in LOCs, effort perspectives depicted here only give a measure of implementation effort and not the engineering and research effort. It is hardly possible to represent the research time spent on each LOC. Sometimes few LOC required considerable research effort to be safely introduced into the framework. This problem is commonly known with very large software system improvements. Therefore, we note that LOC amounts render which part requires extensive modification to support SVP assumptions, SVP language implementation (i.e. $\mu$TC language), SVP architecture implementation (i.e. Microgrid target).

Figure 6.2: Distribution of changes over GCC framework components. The GCC framework comprises 5 major components: front-end (FE), middle-end (ME), back-end (BE), libraries (LIB), infrastructure (INFRA). The changes are made in GCC 4.1 core release which is a special release of the compiler toolchain with one single C front-end, all existing back-ends, and all optimization passes. A main extension is the addition of a new specific Alpha-extended back-end. Most changes in the middle-end are to handle SVP assumptions safely and properly during transformations.

Figure 6.2 shows large changes in GCC’s back-end. A specific back-end supporting SVP instruction has been created and based on the existing Alpha back-end. The back-end generator is extended with methods to support architecture features regarding to SVP operators (part of the abstract machine) added to the abstract machine and synchronized communication channels. For that purpose, peephole and back-end optimizations are extended to permit SVP assumptions. Similarly in the middle-end, inter- and intra-procedural optimizations are extended for SVP assumptions that are exposed in the $\mu$TC language. Generally,
exceptions in optimizations had to be taken care of with special exception handling cases with no need to redesign the entire optimization algorithm. However, some aggressive optimizations have been completely disabled because of the hazardous affects they have on concurrent code. For instance, *instruction reordering*, employed by the *instruction scheduler*, may cause trouble when changing the order of instructions and then break SVP assumptions.

The back-end generator is extended to support SVP actions (properties of the execution model) as native operators. The SVP implementation has the property of extending an existing set of actions (extension of an existing instruction set [43]). Therefore, we apply the same scheme in a way that exposes these properties in the back-end; the back-end gets extended. The *create* action is implemented as a native back-end operator comparable to a special call with specific side-effects. At the create issue point, the flow of control acts differently from being transferred to the routine as in the imperative paradigm. The create operator maps the separation of control flows and allows the back-end optimizations to deal with multiple concurrent control flows. Other SVP actions, such as *break*, *kill*, are also implemented as native operators. Native operators are part of the GCC abstract machine that enables a more generic concept of operations. Thus, it becomes simpler to retarget to another instruction set.

6.3 Compilation challenges

This section collects the experience acquired during the compiler research and development, and the observations while compiling a concurrent program. The challenges encountered are both generic and specific to the Microgrid target: e.g. lack of information about new types or program structures throughout compilation stages and also dragging relevant information about concurrency from one stage to another without any loss.

6.3.1 Issues related to concurrency

The first of the challenges results from the way concurrency is exposed in the SVP concurrent programming paradigm. Integrating concurrent idioms into a modern imperative compiler is one technical contribution to this thesis.

General difficulties

Boehm [48] exposes the limits and dangers of compiler-driven optimizations when using library functions for exposing concurrency in code. However, integrating SVP concepts natively into an existing compiler infrastructure allows one to reuse existing and adaptable optimizations which have been researched and improved for decades. Thus, it allows one to perform more efficient code
generation instead of reimplementing from scratch all existing optimization algorithms. In order to achieve this, the compiler must be made aware of the SVP constructs and semantics and must adapt to these in the code as sequential-based and parallel-extended optimizations. Moreover, the compiler must protect concurrent regions in the code (especially synchronizing objects in $\mu$TC) from aggressive code optimizations.

Besides the well-known issues of compiler code size and compiler code complexity, the major problem encountered is that this sequentially-oriented compiler infrastructure is concurrency-agnostic. Such compilers have been developed over years following an incremental development pattern rather than rebuilding the whole system from scratch when new assumptions or components are inserted. Furthermore, the compiler assumes sequential execution throughout its compilation stages: from optimization algorithms to code generation schemes. Our approach requires concurrency support, not by using plug-ins or external libraries, but by extending the compiler infrastructure, i.e. models presented in [20]. The novelty and hence contribution of this work, in addition to the obvious, i.e. an efficient working compiler for the Microgrid architecture, is in identifying and presenting the challenges encountered while extending a sequentially-oriented compiler.

Parallel assumptions

A bird’s eye view of the integration of the SVP constructs would be the extension of the C sequentially-oriented front-end with the new language constructs which then map into extra instructions and special features in the back-end as the compiler transformations in Section 4.2 illustrate. However looking closer at the compiler infrastructure, it also has to support new assumptions for dealing with the idioms in order to produce valid code. Therefore, new nodes in the tree representation and new objects in the internal representations are added to support parallel semantics throughout the compilation stages. The extension of the infrastructure with new idioms implies the addition of new semantics in the assumptions made in the compilation schemes. Hence, the high-level to low-level representation translations are extended to propagate the SVP semantics.

Sequentially-oriented compilers (in our case, the GCC compiler) make the assumption that only a single thread of control is running at any time. In the $\mu$TC language, the thread function is the smallest composition unit which is seen as a concurrent region with its own context of objects. Thus, a family of threads is a set of concurrent regions which are executed as multiple concurrent control flows running at the same time between the issue point (i.e. create) and the completion point (i.e. sync) in the parent context, as shown in Figure 3.2. Furthermore, communication is allowed between concurrent regions using synchronizing objects. In passing parameters, communication may occur between the issue point and the completion point in the parent thread and all threads using globals and with the first child using shareds. Also synchronized communication may occur between sibling threads using only shared synchroniz-
Chapter 6. Implementing the SVP compiler

ing objects, as shown in Figure 3.6. These objects are expressed in the internal compiler representations with an extra attribute of a given object type. Nevertheless, they cause problems with sequential assumptions in the optimizations if the compiler does not sufficiently distinguish between the synchronizing objects and the regular ones (cf. Chapter 5).

For example in Figure 3.11, a conventional C compiler would optimize away the statement computing the reduction in thread function \texttt{ddot} during dead-code elimination since the code resides in a leaf of the control flow graph (CFG) which never returns anything (i.e. the statement is assumed useless). In SVP, the statement has different semantics since a synchronizing object is present. We also observe that some optimizations are harmless for sequential C code but dangerous for parallel \textmu TC code, as reported in Chapter 5.

Another example in Figure 6.3 uses a synchronizing object as a token across adjacent threads, with C semantics, the statements \texttt{tmp = X}; and \texttt{X = tmp}; would be optimized out with the use of various standard optimizations such as copy-propagation, instruction reordering or combining optimizations. With \textmu TC semantics, the optimized program would not be semantically equivalent to the original program because of broken synchronized communication channels. The \textmu TC compiler is aware of this communication and ensures their validity even in the case of aggressive rearrangements in instruction scheduling. Although those optimizations obey 100\% the C semantics, they can change the meaning of \textmu TC programs and cause visible effects that can break the \textmu TC semantics. The optimizations have been adapted in order to avoid endangering the semantics of \textmu TC code.

6.3.2 Issues related to the Microgrid

As part of the challenges are caused by architecture decisions that forced the compiler to be developed and extended in a specific way.

Resource mapping

In the case of \textmu TC, a new kind of object has been added with different access semantics: \textit{shared} synchronizing objects whose read and write operations have the i-structure semantics described in [62]. These are intended to directly map to registers with i-structure behavior in hardware. Here, we observe that the approach taken by the SVP core architecture implementation creates an issue that we believe has not been researched previously: as a way to maximize the number of threads per core within a limited physical register file, threads can be created with \textit{variable register windows} where a variable number of physical registers are mapped in the architectural register window of each thread for either local (exclusive) use, or physically shared as i-structures with other threads.
Chapter 6. Implementing the SVP compiler

/* thread function definition: 1 shared, 1 global. */
thread void foo(shared double X, double Y)
{
    int tmp = X;
    create(fid;;0;1000;1;;) bar(y);
    sync(fid);
    X = tmp;
}

thread void main(void)
{
    /* point of use: create a family of 1000 threads */
    create(fid;;0;1000;1;;) foo(a = 0, b);
    sync(fid);
    /* after synchronization 'X' contains the result of foo in the parent. */
}

Figure 6.3: Example µTC: Shared object used as a token to enforce sequential constraints.

An example is given in Figure 3.10. This removes the long standing assumption that the size of the architectural register window is a constant for all programs. Also, the compiler cannot reuse physically shared registers via spills: a spill itself would require a read from the register, thus forcing synchronization with the previous thread and reduce concurrency; a write of a local temporary value that motivates a spill would unblock any read in the next thread with the temporary value instead of the final desired dependency. Only local registers, exclusive to each thread, can be used to map temporaries and automatic variables.

As a naive route towards reusing existing register allocators for the C language, we could arbitrarily select a common fixed number of registers for shared and local use by all programs; these numbers would be part of a standard application binary interface that compilers would adhere to when generating code. However, a conservative selection that would both leave sufficient room for sharing between threads and local use by each thread would cause excessive pressure on the register file, by forcing the hardware SVP processes to allocate a large physical register window for every thread created; this in turn would reduce the local concurrency by capping the number of threads that can be simultaneously allocated on each core. As this has a direct impact on the amount of latency the architecture can tolerate on asynchronous operations, we deem it desirable to adapt the architectural register window size to the requirements of each thread function.

The framework where this issue is addressed can be described as follows:
Chapter 6. Implementing the SVP compiler

- the selection of the layout of the visible register window for a given thread function is the responsibility of the compiler; it is provided to the assembler along with the generated code for the thread function;

- given a Microthreaded architecture, this selection is constrained by the architectural limits, namely the maximum number of visible architectural registers per class (e.g. integers/floats) and possible special registers which cannot be shared (e.g. ReadAsZero);

- due to µTC semantics, the selection is further constrained by the arbitrary composability of constructs allowed by C and, by extension, µTC; in particular, the maximum number of registers that can be declared as parameters for a given thread function \( F \) to be shared at run-time with its parent cannot be greater than the maximum number of local registers available for sharing by any thread function that can create a family of \( F \).

We formalize this as follows: considering a single register class on an SVP architecture implementation where there are at most \( R \) visible registers, the compilation of a thread function \( F_p \) and any of its potential child \( F_c \) is constrained by the following formulae:

\[
2S_p + G_p + L_p \leq R \tag{6.1}
\]

\[
S_c + G_c + C_p + L^*_p \leq L_p \tag{6.2}
\]

where \( S_{c/p}, G_{c/p} \) and \( L_{c/p} \) are the number of shared, global and local registers in the register window layout declared to the assembler for \( F_c \) and \( F_p \), \( C_p \) is the number of overhead local registers to perform the create operation, and \( L^*_p \) the number of local registers that must be reserved in \( F_p \) for other uses (e.g. thread local storage pointer). Formula \( 6.1 \) represents the architectural constraint; the number of shared registers is represented twice since there are sharing regions with both adjacent threads, as illustrated in Figure 3.10. Formula \( 6.2 \) represents the sharing of the local register window of a thread with its child family. Both formulae imply a recursive constraint on the entire concurrency tree of a program. In addition to these “hard” constraints, the architecture also suggests optimizing register file usage, by choosing register windows for all thread functions involved in a program such that the total number of physically required registers is kept as small as possible.

This framework is new since an optimal solution to this formalized constraint system is a code generation algorithm where the number of local registers available for code generation is both an input and an output of register allocation. In particular, while the selection of a larger number of local registers in general allows for reduced memory accesses, it is even more important to allow for more concurrency by reducing pressure on the register file.
Register window

Along with the register mapping, the SVP architecture implementation uses variable register classes (global, dependent, local and shared). These register classes are dynamically-sized based on the thread function to which they correspond. One thread function’s context has one register window which differs from one thread function to another. In addition to the previous section, the back-end would require different register conventions for different thread functions. These thread functions are instantiated with the create mechanism which reuses the call mechanism of the sequential model with special extension to handle SVP side-effects on control flows and parameter passing. The calling convention is then not fixed with this register window scheme and the register allocation is severely impacted as previously described. The SVP compiler makes restrictions on fixed register classes since the back-end register allocator appeared to be too complex to realize within the research time frame of this thesis work.

Communication channels

The exposure of concurrent regions and the exposure of inter-thread communication is also a specific issue with the Microgrid target implementation. This issue has been extensively covered in Chapter 5. In theory, communication channels are synchronizing objects added in µTC and mixed with conventional non-synchronizing objects. The synchronizing objects are also present in the Microgrid instruction set with a mapping onto a pair of registers for shared communication channels. Representing communication channels in concurrent programming paradigm is a challenge; SVP implements communication channels on registers. The language and hardware specifications of the implementation of these global and shared communication channels are the source of most aforementioned challenges. The main advantage of this implementation is the inter-thread communication is cheap when using registers and thread family creation is cheap as well.
6.4 Discussion and conclusion

6.4.1 Language and architecture pressure

Compiler research implies the existence of an input language and a target architecture. In the context of the SVP computing system, changes in both language and architecture severely impacted on the compiler. It is difficult to reflect in this thesis the scope of their impact besides the work already described to overcome SVP implementation. It is clear that compiler research is driven by forces coming from both architecture and language which very much pressurize the compiler functionality. A simple change in the instruction set may break internal compiler assumptions in the way it is being processed during compilation. In the case of SVP, for instance, the use of a ‘pull’ mechanism with the create operation in the instruction set made the back-end more complicated than it could have been (cf. Sections 3.3.6 and 4.3.3). The back-end has then to be aware of the bindings between the resources of the argument list and the resources of thread parameters.

The ‘pull’ create mechanism takes the created family’s globals and shareds directly from the context of the parent thread. This ‘pull’ mechanism has two downsides:

- it creates a dependency on the parent thread’s context that is unwanted for creates with no link with their parent, called continuation creates in the SVP model.

- it complicates the compiler design, as the registers that were shared with the child family would be removed from further allocation (frozen) between the create and sync.

On the contrary, a ‘push’ create mechanism would have made the compiler back-end much simpler. The parent thread would explicitly copy the globals and shareds over to the child family’s context, thus removing any dependency on its context. Next to this, there is one dependency left on the parent context: the sync register. This register is cleared by create and written when the family has completed. This meant that the register is still off-limits between create and sync in the program. To remove this last dependency, the create instruction is split into: ‘create’ and ‘sync’. The destination register of the ‘push’ create is cleared as usual but is written when the family’s context has been created and globals and shareds can be written to it. The ‘sync’ instruction writes its destination register when the specified family has completed. Since both these instructions can be considered atomic by the compiler, no special register allocation logic is required between the create and sync point in a program. By having the parent explicitly requesting all interaction (such as sync and parameter passing) with the child family through its family identifier, families no longer need to know who and where their parent is.
Despite platform and input language evolutions, the compiler must be always up-to-date to provide certified and efficient code. Small changes in the Microgrid target definition may damage the compilation process and generate hazardous code. Furthermore, the design of the \( \mu \)TC input language impacted the way in which the compiler research has been conducted. \( \mu \)TC is not an extension of the C language but a sub-set with new constructs and assumptions. Therefore, the C front-end properties must not be used entirely without the insurance of delivering SVP-certified code. With the presence of SVP communication channels, most compiler optimizations need to be certified before use. During this research, some optimizations have been studied and reported in Chapter 5. Evolution in \( \mu \)TC has to be well-advised to prevent any code hazards. Impact of both architecture and language changes may be considerable on the compiler. This thesis, in its scope, reflects the impact of integrating concurrency in compiler modeling and specific Microgrid issues.

### 6.4.2 Support native concurrency

Chapter 4 has depicted the compilation techniques investigated during the compiler development. Changes embedded in the compiler, as presented in Section 4.3 to integrate concurrency in a conventional compiler, have costs for a research team in development overhead. Figure 6.2 stresses a distribution of effort in implementing these changes in the SVP GCC-4.1 compiler. This thesis work presents solutions to support native concurrency into an imperative-language compiler. This chapter has shown the different parts to extend to enable SVP properties in the infrastructure of the compiler: in the front-end, the middle-end, the back-end and the infrastructure itself.

### 6.4.3 Conclusion

The major contribution of this work is the existence of a working SVP compiler. The working compiler development is as close as possible to production quality within the time constraints of this research. Although it is very difficult to obtain a complete verification of the compiler, it is more than enough for a proof-of-concept work. The other contributions are the presentation of the challenges to integrate concurrency into a compiler’s infrastructure, and the areas of danger with SVP compilation. The compiler has room for improvement, mainly because of research time limitations and manpower. The areas where it can be improved are code validation and verification, and also SVP-specific optimizations where it would be interesting to observe what impact they would have on code performance and quality. Limitations on research time prohibited further compiler development enhancement leaving these improvements for future work.

The approach undertaken has been to embed concurrency idioms into a conventional imperative sequential compiler as against an external library approach.
Both approaches have advantages and disadvantages. Our thesis is to expose explicit concurrency to the system infrastructure and to avoid undefined behavior in concurrency. Furthermore, the design choice of extending an existing compiler framework in contrast to writing a compiler from scratch, despite human effort understanding compiler machinery, has reduced the amount of time to get a working compiler within the project duration. The presence of the compiler allows experiments with a set of non-trivial larger benchmarks instead of simply having hand-coded versions. This chapter has exposed challenges in existing compiler assumptions when dealing with the programming of multicore architectures.