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.
The previous chapter introduced the SVP Execution Model and its hardware and software implementations. The CSA group aims to build a fully parallel computing system where the Self-Adaptive Virtual Processor execution model is the cement adhering the components. As Figure 3.1 explicitly illustrates, the SVP system consists of two implementations which are bridged via an SVP-aware compiler. In practice, a program utilizing the $\mu$TC language implementation is translated into a corresponding machine-level representation targeting the Microgrid hardware implementation.

In this chapter, we first introduce the reader to the idiosyncrasies of program transformations in an abstract manner. We then discuss the compilation schemes that are performed in order to bridge the $\mu$TC language implementation and the Microgrid hardware implementation. To explain this, we also use a theoretical approach to address the description of compiling a concurrency-oriented language for a multicore architecture.

The main contribution to the reader is to present the major abstract steps of program compilation. New and unconventional computer architectures are easy to envision. Nonetheless, programming these architecture is an entirely different matter. This research aims to prove that the SVP architecture implementation can be targeted with an SVP-extended modern imperative compiler.

The contents of this chapter are based on this publication:

Hence, we will set up the concurrent context in opposition to conventional assumptions. Moreover, the reader will get a clear understanding of the transformations performed by our compiler displayed in Figure 3.1. We will also stress the conceptual issues that can endanger the concurrent code.

4.1 Basics in compiler transformations

4.1.1 A compiler is a transformer

Compilers are major computer programs in computer science because of their central position as shown in Figure 1.6. The reason why this component is significant is because it is the only bridge between two worlds: the Software world and the Hardware world. They are enabling technology allowing the software layers to work with the hardware layers. In practice, the main purpose of a compiler is to transform input code into a semantically-equivalent output code without any semantic loss. The input code written in a computer language, referred as the source language, is transformed into another computer language, named target language. In most cases, the source language is a high-level language, e.g. Java, C, etc.; it is transformed into the target language which is lower-level than the source language, e.g. a machine language, series of machine instructions.

\[ T(X) \Rightarrow X' \] (4.1)

A compiler can be seen as a transformer from an abstract perspective. Formula 4.1 represents a transformer \( T \) which takes a program \( X \). Then, the result of this transformation is the semantically-equivalent program \( X' \). A simple analogy is to a natural language translator who conceptually performs the same tasks by translating a language he/she hears (or reads) into another one. The translator gives results in a spoken or written form.

Moreover, a compiler is also capable of producing a more-optimized output program. Its purpose then is not only to generate correctly transformed programs, but also an optimized program to be run more efficiently on the target machine or compiled into a smaller size if the target machine is an embedded system device. Again with our translator analogy, he can evaluate the input form in order to render a simpler translation with a smaller amount of words to express the same meaning or even to reproduce the same semantics with non-complex words.

Formula 4.2 represents an advanced transformer \( T \) able to generate an optimized output program \( X^* \). An extra input \( a \) may be required to select a set of parameters to control specific optimizations of the optimizing transformer. We can then command the transformer to give us multiple representations from the same input program; the translator would do similar if we tell him to only use layman’s words.
The major concern in translating is to preserve the semantics of the input program. During a G20-summit, we do not want the translator to misinterpret the speech about nuclear-weapon proliferation agreements for instance. Therefore extreme care must be taken in ways of transforming an input. Consequently, the same applies to an optimized output when we want to obtain a more-concise program as a result. However, we do not want to lose any semantics from the input program. Throughout any program optimization stage, the program’s semantics must remain the same as the semantics of the input program $X$.

### 4.1.2 Different types of compilers

Multiple kinds of compilers can be distinguished and derived from the first definition. Here is a list of the most commonly known

- **A source-to-source compiler** is a type of compiler that takes a high-level language as its input and outputs a high-level language. For example, an automatic parallelizing compiler will frequently take a high-level language program as input and then transform the code with the addition of parallel code annotations (e.g. OpenMP).

- **A compiler-compiler** is a compiler generator tool that creates a parser, interpreter, or compiler from some form of formal description. For instance, a parser generator, whose input is a grammar (usually in BNF) of a programming language, and whose generated output is the source code of a parser.

- **A just-in-time compiler** first translates applications into an intermediate representation known as bytecode. This bytecode compilation is prior to execution. Bytecode is not the machine code for any particular computer, and may be portable among computer architectures. This is then compiled to native machine code (i.e. dynamic compilation) when required, at run-time prior to executing the code natively, e.g. Java compiler.

We have introduced the different types of compiler for the sake of the reader who needs to know that *compiler* is a vague word in the broad computing world. In this thesis, when we refer to compiler, we assume the most common use of a compiler which is the translation from a high-level language (e.g. $\mu$TC) into a low-level representation (e.g. assembly code for the Microgrid).

---

1The reader must be aware of the fact that this list of compiler types is not mutually exclusive.
4.1.3 The principles of a compiler

Compilers are engineered objects and very large software systems built for specific transformation purposes. Building a compiler requires myriad design decisions which will be explained throughout this chapter and Chapter 6, each of which has an impact on the resulting compiler. There are two major principles of compilation that should not be compromised though during a compiler’s construction.

The compiler must preserve the semantics of the input program being compiled.

This first principle is fundamental in any transformation. The compiler must not lose any meaning of its input program. Correctness is then a priority in compiler transformation and it will be discussed later in the experimental evaluation of the compiler in Chapter 7. The second principle that a compiler must observe is more pragmatic.

The compiler must improve the input program in some perceptible way (in the case of an optimizing compiler).

A conventional compiler improves upon the input program in different ways depending on the targeted machine. The improvement of a program is subject to interpretation in its meaning. For instance, in the environment of embedded systems, a low usage of energy-expensive operations is a priority. Therefore, the compiler will utilize in priority the least energy-consuming operations possible for a specific task. In general, a compiler also tries to optimize the memory usage of a program by using the on-chip memory as much as possible (which is cheaper in access time but present in smaller quantities than off-chip memory). Even if, on some architectures, it is possible to run a program by only using off-chip memory. However, the program executed would be much slower in terms of execution cycles. Consequently, the improvement constraints in that case are the lowest possible execution time and the lowest use of time-costly off-chip memory.

To summarize this section, we only presented what is necessary to understand the context of compiler transformations. The reader now has the prerequisites to comprehend the scope of the next section discussing the compilation schemes of the SVP compiler. In Section 4.3, we will explain in further details how modern compilers are built in a more technical way. For now, in the following section, we keep an abstract perspective on how the compilation is being made. We want the reader to grasp in what context these transformations must occur and what are the requirements for correct transformations.
4.2 SVP compilation schemes

In this section, we express SVP compilation transformations with a simple formalism. These SVP compilation transformations bridge the SVP language implementation (i.e. the \( \mu TC \) language) into the SVP hardware implementation (i.e. the Microgrids architecture). In the state of the art, these compilation transformations are most commonly called ‘compilation schemes’. These describe the properties of a compilation process. Here the basic compilation scheme \( T \) is shown in Figure 4.1. Furthermore, this transformation rule \( T \) is an abstract representation of the SVP compiler. All the transformation rules described in this section are extensions of the regular C compilation schemes.

\[
T \left[ \text{code} \right] \Rightarrow \text{instructions}
\]

**Figure 4.1:** Basic compilation scheme \( T \) takes a section of code as input on the left-hand side. Then, it gets translated into series of instructions and directives, on the right-hand side.

SVP compilation is performed on each \( \mu TC \) thread function. In \( \mu TC \), all functions are thread functions; regular C function calls are treated in a special way which is explained at the end of this section. Moreover, we consider a thread function as a procedure. The compilation processes apply on a program which comprises one or more procedures. As shown in Figure 4.2, a thread function is processed through the compilation scheme \( T \) as a whole.

\[
\begin{align*}
T \left[ \begin{align*}
\text{thread} & \quad \text{break_type} \\
\text{thread_name} & \quad (\text{thread_args}) \\
\{ & \\
\text{Body} & \\
\} 
\end{align*}\right] \Rightarrow \left\{ \begin{align*}
.\text{asm} & \quad \text{thread_name} \\
.\text{registers} & \quad \text{gi, si, li} \quad \text{gf, sf, lf} \\
\text{thread_name} : & \\
T[\text{Body}][E[\text{thread_args}]] & \\
\text{end} & 
\end{align*}\right\}
\end{align*}
\]

**Figure 4.2:** Compilation scheme \( T \) for a thread function. Here, we do not yet consider the special treatment of the thread arguments (i.e. compilation scheme \( E \)). This is further expounded in Figure 4.3. The result of the transformation \( T \) is the corresponding assembly procedure (i.e. starting with “.asm \text{thread_name}”, finishing with “\text{end}”) on the right-hand side.

A thread function is transformed by the SVP compiler into a sequence of instructions and assembler directives, as the compilation scheme in Figure 4.2 illustrates. On the right-hand side of this figure, the assembler directive .registers is responsible of defining the register windows where “gi, si, li” are compiler-calculated numbers of registers for global, shared and local classes for the integer thread context \( \text{thread_name} \) and respectively “gf, sf, lf” for the floating-point thread context. The \text{end} instruction is the exit point of the concurrent region. If \( \text{Body} \) does not have extra \( \mu TC \) constructs then regular C compilation schemes
apply.

Figure 4.3 also illustrates the compilation of a *thread function* where any of its parameters uses a synchronized communication channel declared as *shared* or *global* in the $\mu$TC program. As discussed in Section 3.2.2 and Section 3.4.5 the channels are handled with care by the SVP compiler to preserve their semantics. The *global* variables are read-only in the scope *Body*. The SVP compiler does not assign any *global* variables in the produced code. At thread creation or when data becomes available in parent’s context, these variables are implicitly initialized by the architecture. Moreover, all statements held in *Body* which are using variables declared as *shared* are preserved in the transformed program shown on the right-hand side of the figure. There is a distinction made by the compiler when a $\mu$TC *shared* variable is accessed: marked in the compilation scheme with accessor methods, *write*(*id* _name_) and *read*(*id* _name_) respectively mapped in *shared* and *dependent* register classes.

Any *thread function* may create a subordinate family. The compilation scheme in Figure 4.4 shows the SVP *create* action on the left-hand side and the transformed output after compiler transformation on the right-hand side. The first component is the initialization of thread arguments in the parent context via $T[thread\_args]$. The *allocate* instruction holds offsets of the parent context for parameter passing to the child context. “$\epsilon(w),\epsilon(x)$” are compiler-calculated offsets in the parent context for the first passed *global* and first passed *shared* for integer variables and respectively “$\epsilon(y),\epsilon(z)$” for floating-point variables as explained in [61]. Then, the *set*-like instructions set up the configuration of the family of threads to be created. The *create* instruction is responsible for the generation of the family of threads *thread_name*. 
where $E$ is defined as,

$$E[\text{type id}_\text{name}, \text{Rest}] \Rightarrow E[\text{Rest}]$$

$$E[\text{shared type id}_\text{name}, \text{Rest}] \Rightarrow \text{id}_\text{name} \cup E[\text{Rest}]$$

$$E[\text{type id}_\text{name}] \Rightarrow \emptyset$$

$$E[\text{shared type id}_\text{name}] \Rightarrow \text{id}_\text{name}$$

and $F$ as,

$$T[\text{id}_\text{name}_1 = \text{id}_\text{name}_2; \, \text{Rest}][\text{shared set}] \Rightarrow \begin{cases} 
\text{write(}\text{id}_\text{name}_1) = \text{id}_\text{name}_2; & \text{where } \text{id}_\text{name}_1 \in \{\text{shared set}\}, \\
T[\text{Rest}][\text{shared set}] & \text{id}_\text{name}_2 \notin \{\text{shared set}\}. 
\end{cases}$$

$$T[\text{shared set}] \Rightarrow \begin{cases} 
\text{id}_\text{name}_1 = \text{read(}\text{id}_\text{name}_2); & \text{where } \text{id}_\text{name}_2 \in \{\text{shared set}\}, \\
T[\text{Rest}][\text{shared set}] & \text{id}_\text{name}_1 \notin \{\text{shared set}\}. 
\end{cases}$$

$$T[\text{shared set}] \Rightarrow \begin{cases} 
\text{write(}\text{id}_\text{name}_1) = \text{read(}\text{id}_\text{name}_2); & \text{where } \text{id}_\text{name}_1, \text{id}_\text{name}_2 \in \{\text{shared set}\}. \\
T[\text{Rest}][\text{shared set}] 
\end{cases}$$

$$T[\text{shared set}][\text{shared set}] \Rightarrow \begin{cases} 
\text{id}_\text{name}_1 = \text{id}_\text{name}_2; \\
T[\text{Rest}][\text{shared set}] & \text{otherwise.} 
\end{cases}$$

Figure 4.3: Compilation scheme $T$ for a thread function where arguments have to be determined whether or not there are potential synchronizing objects declared as shared variables in the $\mu$TC program (i.e. used as SVP synchronized communication channels). The compilation scheme $E$ determines whether thread arguments are shared variables or not and returns a potentially empty set of shared variables. This set is then used in combination with the compilation scheme $F$ for dealing with synchronizing variables used in the statements of Body. The compilation scheme $F$ assumes that the code has been flattened. Moreover, the variable identifiers $\text{id}_\text{name}_1$ and $\text{id}_\text{name}_2$ can refer to the same variable and are then distinguished by the index number. write($x$) and read($x$) are not function calls but accessor methods on a synchronizing object $x$. 

65
A thread function can be defined as breakable by using a break_type other than void and by using the µTC break construct in its Body and is transformed as shown on Figure 4.5. This transformation involves the evaluation of the value of expression expr. The type of this value is then compared to the break_type. This value will be collected in the parent’s context as shown in Figure 4.4 with the object defined with the setbreak instruction. Moreover, an extra exit edge is added to model a new potential exit point in the execution path of the program.

Figure 4.4: Compilation scheme for µTC create action. There are three parts to distinguish in the right side: first the initialization of the thread arguments with $T[\text{thread\_args}]$; second the family settings of the to-be-created family with the set-like instructions; third the create instruction.

$T[\text{break}(\text{expr});] \Rightarrow \text{break } T[\text{expr}]$

Figure 4.5: Compilation scheme for µTC break action. The corresponding thread function is defined as breakable by the presence of a break_type other than void.

The µTC language allows the use of regular C function calls. Nevertheless, the compiler transforms them as shown in Figure 4.6. On the right side of the figure (i.e. the parent context), instead of conventional caller code, the compiler produces a family of one thread targeting a call_gate and generates a family identifier. As shown in Figure 4.7 the call_container function_name (function_args) is generated and wrapped around the conventional sequential generated caller code in a separated context, call_gate. This allows the proper use of sequential functions in thread functions. The callee generated code is conventional sequential code.
4.3 Under the hood of SVP compilation

In the previous section, we have looked at the compilation schemes that the SVP compiler performs with a formalism which reflects the requirements of the SVP compilation process. Section 4.1 already introduced the basics of transformations. In this section, we look under the hood of the compilation process to see, still from a theoretical standpoint, the steps of compilation. SVP compilation schemes extend standard C compilation schemes. We found inspiration for this section with knowledge from thorough compiler and compilation books such as [72, 73, 74].

4.3.1 Overview of compiler structure

Theoretically, a compiler is a single-box transformer as illustrated in Figure 4.8. We refer to the input as the source program and the output as the target program. The black-box transformation corresponds to Formula 4.1.

A compiler must both understand the source program presented for compilation and map its properties to the target machine. These distinct tasks suggest a division of labor and an explicit design within the black box. The compilation process is decomposed into two major pieces: the front-end and the back-end as illustrated in Figure 4.9.
A ‘simple’ compiler consists of a language-dedicated front-end that ‘understands’ the source-language program and a specific back-end that maps the program to the target machine. These two pieces are combined inside an infrastructure which drives the compilation process and provides data structure, tree and graph structure representations, etc. The front-end emits an internal version of the program, called Intermediate Representation and presented as IR in Figure 4.9. At any point in the compilation process, the compiler has an IR of the input program which will be then converted into the target machine code during the last stage of compilation.

We described here a simple two-stage compiler. The first stage (i.e. the front-end) ensures that the source program is well-formed and it converts it into the IR. The second stage (i.e. the back-end) maps the IR program into the instruction set and the finite resources of the target machine. In reality, compilers are more complex with multiple front-ends and back-ends. A front-end is dedicated to a language and the back-end is specific to a target machine with its forms of instructions and its memory structure. Therefore, having a language-independent and machine-independent IR gives many more possibilities of reusing and combining the different front-ends with various back-ends.

In Section 4.1, we have seen that one of the principles of compilation is to improve the input code. The two-stage compiler design is thus inappropriate and then evolves into an optimizing three-stage compiler as shown in Figure 4.10. The optimizer stage is an IR-to-IR transformer and performs one or several passes over it before emitting an optimized IR. Formula 4.2 represents an abstract vision of this optimizer. The compiler then has different levels of IR throughout the compilation process. The IR evolves accordingly the functionality of passes. Finally, the back-end deals with an optimized IR when emitting the target program.
Figure 4.10: A classic optimizing compiler is formed with three distinct stages. An optimizer is therefore added in between the front-end and the back-end. This optimizer stage is often referred as middle-end.

The optimizer can make one or more passes over the IR, analyze the IR, and rewrite the IR. The optimizer may rewrite the IR in order to produce a faster target program or a smaller program from the back-ends. Each pass of the optimizer has an objective and can be iterated until reaching the optimal solution of its functionality.

Conceptually, in a three-stage compiler design, there is a clear separation of concerns inside the compilation process:

- the front-end’s concern is to understand the source program, to report the lexical, syntax, semantics errors, and to build the first IR form.

- the middle-end’s concern is to improve the IR form as much as possible, this is being done independently from the source language and from the target language.

- the back-end’s concern is to map the optimized IR form onto a bounded set of resources of the target machine in a way that aims to make efficient use of target resources.

In practice, modern compilers are based on a three-stage design with a complex infrastructure to allow combinations between a vast set of target machines, a large amount of optimization passes, and a wide set of source languages. The infrastructure provides a sufficient amount of control commands to the user to take advantage of the compiler features (e.g. abstracted in Figure 4.12 a set of parameters to control specific optimizations and commands $\alpha$). The infrastructure also provides generic symbol tables, tree and graph representations, generic and reusable methods (i.e. cross-platform compatible).

Figure 4.11 represents an advanced compiler design which we use as our research framework. We now look closer at the compilation stages. Of these three stages, the middle-end has issues and problems that arise while optimizing a program and it will be discussed in Chapter.
4.3.2 Classic Compilation work-flow

Within the scope of explanation for an imperative-based compiler, we now look at the compilation process itself. Represented as a work-flow in Figure 4.12, the compilation process is a sequence of stages performed inside the compiler. A stage is also defined by a series of passes. A pass is either an IR analysis, or an IR optimization, or an IR writing. A source program being compiled goes through all stages and various passes. Therefore the compilation process becomes opaque and complex to the compiler user. However, the user can also use control commands to record the different IRs throughout the compilation process. Usually, these IRs do not contain all the information carried through the different stages mainly because they contain too much information to display. Moreover, the IR form is not trivial to understand (depending on the stage where it is been recorded). This section introduces the different stages and passes of a classic compilation using a work-flow perspective.

The Front-end

The source program is first analyzed by the front-end of the compiler. The front-end is at the first stage of the compilation. It gives most of the feedback to the user concerning whether his program is well-formed. This is done during a first pass on the source program scanning the stream of characters used in defining the program’s code. After scanning, a stream of tokens is then generated from the input. The scanner is often referred as lexical analyzer. It aggregates symbols to form tokens and applies a set of predefined rules from the source language definition.
Figure 4.12: A work-flow representation of a compilation process. This abstracts a modern optimizing three-stage compiler design. The three stages are represented with containers; the source program comes as input and the target program is the output of the work-flow. Each square block (block in container) defines a task in the work-flow and interacts (as a pass) with the program’s representation during compilation.
After the lexical analysis, the stream of tokens is sent through the parser of the language-specific front-end. The parser determines the type of input tokens. For instance, if the token is a keyword of the source language or if the token is a regular expression. The parser recognizes the syntax of the source language and provides feedback to the compiler user about the problems with syntactic errors made in the source program. The parser’s output is a concrete model of the source program to be used by the later stages of the compiler. A token is now associated with a class and a location within the program. The parser then keeps track of where the tokens are found while parsing.

Keeping in mind that the front-end performs extended analysis on the source program validity, the front-end contains a Context-Sensitive Analysis (CSA). It generates a large database concerning the details of the program. By checking how values are represented, verifying how values flow between variables, analyzing external component used in the source program, the context-sensitive analysis understands the structure of the computation. The type-checking analysis is important to ensure that there is no problem related to wrong type flow between variables where values flow. All this information is extracted from the source code and then transformed into an abstract tree representation. This is the first Intermediate Representation (IR) of the program which is the output of the front-end. Moreover, the front-end provides feedback on program irregularities resulting from invalid keywords, invalid regular expression, invalid types, etc, along with diagnostic information that the compiler user can use for refining and correcting the input program.

The Middle-end

The middle-end starts when the Abstract Syntax Tree (AST) has been generated from the front-end. This tree-based IR contains information about the code being analyzed and translated. This IR is independent from the source language and is the abstract program representation inside the middle-end. This then allows the middle-end to be generic and therefore reusable for different front-ends. Moreover, the IR used in the middle-end contains no information about the target language which hence provides an abstraction layer to get a target-independent IR. Thus, the middle-end can also be reusable for multiple back-ends. The passes in the middle-end consume the IR, analyze the IR and rewrite the IR.

The first pass of the middle-end generates a three-address IR (also called Static-Single Assignment (SSA)). In this representation, most operations have the form \( x \leftarrow y \ op \ z \), with an operator \( op \), two operands \( y \) and \( z \) and one result \( x \). Some operators, such as an immediate load and a jump, need fewer arguments. Sometimes, an operation with more than three addresses is needed. A new set of compiler-generated variables are introduced in the code and reveal new opportunities to improve the code on a simpler IR. The reason for this is that a variable is only assigned once within the same scope; it therefore permits more aggressive optimizations.
Furthermore, at the beginning of the middle-end, another pass is derived from the tree IR to produce a Control-Flow Graph (CFG). The CFG models the flow of control of the source program. The atomic part of a CFG is called a basic block. A basic block is a sequence of operations that execute together. Control always enters a basic block at its first operation (e.g. a target from a jump) and exits at its last operation (e.g. a jump operation). Edges are used to expose possible transfer of control from one block to another one. The CFG is used in program analysis and by optimizations. As any part of the IR, the CFG can be rewritten by any compiler pass.

On top of the various IRs (i.e. AST, CFG, SSA), the compiler has other internal data structures which are used during the compilation passes. The symbol table is one of them and is an integral part of the compiler’s IR. The compiler encounters a wide variety of objects during compilation – variables, defined constants, procedures, functions, labels, structures and files. As mentioned before, the compiler also generates many objects. The symbol table lists all the program’s objects with information related to their type, their location in the code, their scope, their dimension (for arrays), their field (for records or structures), the number of parameters and their types (for functions), etc. These IRs are used by program analysis passes in the middle-end. A source program is composed of one or more functions. Within the compiler, a function is abstracted as a procedure. Advanced program analysis involves inter-procedural analysis which gives feedback on communicating values between procedures for instance. Here, the compiler analyzes which procedure call is related to a specific procedure. A typing-check is also performed to verify whether a call argument has the proper type for a specific procedure argument. Moreover, procedure calls are analyzed to produce a call graph which gives the nested level of a procedure from a program root perspective (in other words, the starting point).

Using the IRs created by previous passes, the optimizations can be more accurate with sharper information. For instance, the Dead-Code Elimination (DCE) uses the CFG to evaluate which branch of the graph can never be reached by any flow. Then the DCE simply removes the useless branch from the graph. The result of such a pass is really interesting since it can be used to reduce the code size of a given program. Another interesting pass called Common Subexpression Elimination (CSE) searches for identical expressions, evaluating to the same value and analyzes whether it is worthwhile replacing them with a single variable holding the computed value. In later stages, fewer operations will be generated and hence the target program will be smaller in size. Constant propagation looks for constant expressions which values are known at compile time. These constant expressions will then be substituted by their value. The main optimization done here is to produce fewer operations for a similar result. The target program produced will be composed of fewer operations.
Chapter 4. From basics to advanced SVP compilation

The Back-end

The limits of the middle-end and the back-end differ from one compiler to another. In modern compiler design, the back-end is not only responsible for generating the target program and to map to a limited amount of physical resources. The back-end has evolved to an optimizing back-end nowadays. Further optimizations can be done before code emission (i.e. the last stage of compilation) such as Peephole Optimizations. These machine-dependent optimizations typically target small segments of code represented in the low-level IR (i.e. IR close to machine code). It recognizes sets of instructions that do not actually do anything, or that can be replaced by a leaner set of instructions. On some architectures, there is a special addition instruction for large words which allows with one single instruction an addition with more than two operands. Moreover, some series of instructions can be simply substituted by faster ones; this is called Strength Reduction. These optimizations operate on a low-level IR which has been lowered when entering the back-end. Generally, this low-level IR almost looks like machine code with pseudo-instructions using pseudo-registers (or virtual registers which are non-architectural registers not yet register-allocated). Modern back-ends analyze the IR, if not done yet at the middle-end stage, to extract dataflow information. Some modern compilers might reperform analysis on the source program being compiled such as Data-Flow Analysis (DFA). Since the representation of the program evolves throughout compilation stages, the compiler passes require accurate information about the program and therefore frequent updates of the IRs. DFA is a static code analysis (i.e. performed at compile time) and investigates the whereabouts of values in a program. Observing how values flow, the compiler can know exactly how the code would behave at execution time. DFA is a necessity for optimal register allocation and other data-related optimizations.

The last stage of compilation is one of the most important and often the one which makes the difference in terms of performance between compilers: code generation. The code generation is separated in three components:

1. **Instruction Selection**, by definition, chooses the proper instruction in the targeted instruction set for the selected operation.

2. **Instruction Scheduling** results in ordering the selected instructions. This scheduling is a speed optimization which has a critical effect on pipelined machines. Therefore, the instructions can be reordered to improve performance or hide latency based on heuristics of the targeted machine.

3. **Register Allocation** (RA) allocates variables of the program into hardware registers. The main concern is that the program can have an enormous amount of variables to map on a restricted amount of physical registers on the target machine. Based on heuristics, the RA optimizes as much as possible the mapping of the variables to hardware registers, for instance, by putting in hard registers the most-often-used variables and then reduce the use of memory accesses (i.e. stack or other memory structures).
Hardware registers have the quickest read-and-write access time in opposition to other architectural memory structures.

The last stage of compilation is code emission which produces the final optimized representation of the program into the target program. To accomplish all these stages, the compiler uses a considerable number of data structures to represent the details of the input program. Modern compilers are usually “reconfigurable” frameworks which provide a set of generic cross-compatible methods and processes for different source languages and target architectures. The compiler framework provides:

- graph structures,
- tree structures,
- symbol tables or hash tables,
- singly-, doubly-, and multiply-linked list methods,
- graphical representations,
- diagnostic information,
- the infrastructure.

This section has explained how the classic compilation work-flow is a complicated and monumental process. Despite that, the compiler takes special care to follow the two main principles: program semantics preservation and program optimizations.

### 4.3.3 Compilation work-flow of the $\mu$TC language

After describing the classic compilation process (i.e. work-flow of stages) and the SVP compilation (in Section 4.2), we look now at the correlation between compilation of the SVP language implementation and conventional compilation. For the moment, we use an abstract level of representation to help the reader to grasp the essence of SVP compilation. The SVP compilation is comparable to a work-flow which takes as input a source program using the $\mu$TC language and results in producing its corresponding representation using the Microthreaded Instruction Set as target language. The SVP compilation process extends, in practice, the standard C compilation schemes.

**Composition of a $\mu$TC program**

First, we have to observe how a given concurrency-oriented program is built from a functional perspective (i.e distinguish the functional blocks, the communication between the blocks). Indeed the source program, coded in $\mu$TC, is
composed of computational blocks which correspond to thread functions as illustrated in Figure 4.13. From a model view, a thread function is a concurrent region. A µTC program consists of a hierarchy of threads grouped by families in which the hierarchy is exposed with the concurrency tree. This concurrency tree hence shows the concurrent regions of a program and their relationships.

Figure 4.13: A µTC program is composed of concurrent regions on the left-hand side. A thread function is a concurrent region as represented on the right-hand side. And, this thread function defines a family thread job. This family of threads is created by the parent with the create action. For each thread of the family, a thread context is instantiated and contains its own context of variables. The context is constrained by dataflow dependencies. This figure does not represent any inter-thread dependencies.

Families of threads

A thread family encapsulates a group of threads at execution time. However, at compile time, a thread family’s work corresponds to a thread function. The terminology of a thread function being compiled is called a thread procedure. In the execution model, a thread procedure is the atomic concurrent region and it has its own context of variables. The communication channels are explicitly exposed in the program and they represent the dependencies between threads (i.e. with the parent thread, the child thread, the adjacent threads). In practice, the compiler separately processes a thread function from the rest of the program. Nevertheless, once a complete scan of the program is performed, the compiler has a global view of the source program with the IRs. During SVP compilation, a µTC source program is mapped as a sequence of thread procedures. In Figure 4.13, a thread function in µTC is a thread procedure within the compiler and becomes at execution time a concurrent region. At execution time, a thread family is composed of heterogeneous threads which have their own private context. Nevertheless, these threads are statically homogeneous in their µTC code description.
A proper internal representation of a $\mu$TC program

A thread family therefore can have multiple live contexts at execution time. Each indexed thread of this family has its own flow of control. The inter-context dependencies are exposed with synchronized communication channels. Consequently, after decoupling the $\mu$TC source program into a composition of thread procedures, the compiler must preserve their hierarchy in order to preserve the semantics of the source program. For that purpose, the compiler builds the concurrency tree on the same scheme as a call graph in the sequential paradigm. This concurrency tree can be seen also as a creation graph illustrated in Figure 4.14. Note that the creation graph is context-sensitive, each concurrent region has its own context and inter-context dependency information is annotated. Thread procedures are interconnected with the create action. As a comparison with the sequential model, a program is composed of routines that are interconnected with calls. A routine or sub-routine is, by definition, “a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code”. Hence, within the concurrent model, thread functions are the corresponding concurrent routines.

![Figure 4.14: Creation graph example of a given $\mu$TC program. Similar in appearance to the call graph, the creation graph is a representation of the concurrency tree of the program. The creation graph also contains creation relationships between concurrent regions (i.e thread function) as thread parameters in the $\mu$TC program. The depth is the level of encapsulation of a create from the root of the program (i.e. main thread).]

A peek at optimizations

The compiler, at the middle-end stage, performs an advanced inter-procedural analysis to verify which create action is related to which thread function. Since the creator of a thread function is identified, type checking of the thread parameters can be performed as well. Conventional intra-procedural optimizations are performed on each thread function with respect to the synchronized communication channels. This will be discussed in Chapter 5. Figure 4.15 represents the relationships of a concurrent regions with surrounding regions; these relationships must be exposed to the compiler to convey SVP assumptions in compiler
optimizations. The rest of the compilation stages take action on the thread procedure being-compiled such as register allocation and the other passes in code generation. The resulting code representation of the source program is emitted. The output of the SVP compilation is a series of Microthreaded assembly procedures. As a non-concurrency-aware programmer, the resulting target program resembles to series of operations similar in the form to sequential procedures.

**Internal representations: augmenting the conventional CFG**

The concurrency tree is visible to the SVP compiler; therefore, the compiler is able to perform several actions on it: read it, write it and transform it. Potential optimizations can be done to restructure the tree shape in order to enhance the concurrency in a better way than the one exposed. The compiler here performs the second principle of compilation we mentioned earlier: optimizing the source program. To perform such optimizations and further compilation stages, the compiler also requires an accurate CFG. The CFG indeed needs to be aware of the concurrent assumptions related to the concurrent regions and their relationships (cf. Figure 4.15). Moreover, the compiler has to mark a create action and its related sync action. The control flow ‘leaves’ the concurrent region at the create region and eventually comes back at its related sync action. In practice, there is a difference here from a sequential paradigm where a call generates a branch in the control flow from the current context. While branching to the callee and until the end of its execution, the control flow is away from the caller’s code. In our case, a separate flow of control is generated at the create action while the creator’s flow of control continues after the create action as illustrated in Figure 4.16.

The conventional CFG does not make the assumption of multiple concurrent flows of control; therefore, it requires special attention and must be extended properly to support such assumptions as illustrated in Figure 4.17. The CFG exposes the concurrent regions of a program and also the create and sync actions. The SVP compiler is then aware of which areas of the procedure may have more than one control flow. The sync action joins all of the flows of control from the child threads and the parent thread into the parent context. Therefore, after the sync action, the variables used between the child family and the parent are consistent.

**Internal representations: updating the conventional DFG**

While the concurrency tree contains information related to inter-region dependencies, the CFG contains much finer-grained information within the concurrent region. Indeed, the basic blocks composing the CFG map finer behavior inside thread functions. Therefore, inter-context data relationships with the parent or/and with the potential child family (if this is the creating basic block) and adjacent thread are visible to the SVP compiler. The scope of variables within a concurrent region is different from sequential assumptions, when the variables
Chapter 4. From basics to advanced SVP compilation

Figure 4.15: The relationship of a concurrent region with adjacent regions and its parent. Parent P creates a family C of 3 threads which has two parameters: a global \( x_g \) and a shared \( y_s \). The tokens \( G \) and \( S \) respectively represent their flow (by the arcs as ‘read’ \( R \) and as ‘write’ \( W \)) through contexts of concurrent regions. The shared and global channels are marked in the internal code representation, and thus are handled with care.
Chapter 4. From basics to advanced SVP compilation

Figure 4.16: Comparison of the control flows of a sequential call on the left-hand side and a concurrent create on the right-hand side. In the sequential paradigm, when a call occurs, the control flow leaves the caller’s context and goes into the callee’s until completion of the callee’s job. Then, the control flow comes to the operation immediately after the call operation (i.e. at line L+1). Nonetheless, in the SVP concurrent paradigm, when a create takes place, the control flow separates into two: one flow goes on in the parent’s context and the other flow spawns the children threads. There, this control flow separates again in each child thread. At completion (i.e. sync operation at line L+i), the control flows join and after this operation, only one flow remains in the parent’s context.

Figure 4.17: CFG representation of an SVP create-sync block. There is one entry point to this block at the create operation and one exit point after the sync operation. The representation of the control flow separation is shown by two outgoing arcs from the create operation. The Thread Function (i.e. TF in the figure) is an external block and contains the CFG of the job to perform. At the sync, all the incoming arcs converge; subsequently, only one outgoing arc remains.

are defined as synchronized communication channels. Through the DFA, the SVP compiler isolates the channels when realizing the Data-Flow Graph (DFG) of the source program. In SVP, the shared communication channel is defined as one single object in the source program which maps into a pair of objects: incoming and outgoing objects. In order to have an appropriate RA and optimizations, the DFG contains this information with the proper representation of
communication channels. The same works with marking the global communication channel as read-only channel. Thus, the DFG is aware of the read-and-write accesses on SVP communication channels, as exposed in Figure 4.18.

```
thread void foo(shared double a) {
    ...
    t = a;
    ...
    a = ;
}
```

**Figure 4.18**: On the left-hand side, an SVP thread function using one single SVP shared synchronized communication channel. On the right-hand side, its corresponding DFG.

The CFG and DFG of the source program are particularly important when they occur in inter- and intra-procedural optimizations. Any operation from a concurrent region may have a side-effect on another operation in another concurrent region. The SSA form, in the middle-end, is a good example of 3-address operations where the DFG is required to have information related to their inter-context data relationships.

**Special SVP operators**

The back-end of a compiler framework comprises two separate parts: the first is the back-end generator which is generic for any target platform; the second contains descriptions of target architectures about their instruction set, their memory structure, their register structure, etc. The back-end generator employs operator or native operators which correspond to operations in an abstract target machine. This abstract target machine is an abstraction of target platforms with adjustable and generic instruction formats. In the context of SVP compilation, SVP operators extend this abstract target machine to natively implement the SVP actions. From the front-end, the SVP constructs are carried out and lowered down to SVP operators such as create, sync, kill, break operators. Doing this allows the possibility to retarget to another target platform with SVP properties. Moreover, the target description is also extended to map these SVP operators onto SVP instructions.

The create construct in µTC splits into three blocks in the back-end:

- a first block for the arguments of the thread function,
- a second block for the settings of the create and the to-be-created family,
Chapter 4. From basics to advanced SVP compilation

- a third for the operators for generating the create.

The create implements a ‘special’ call with side-effects on the arguments used in the thread function argument list and on the control flows of the neighboring operations (in the execution order). The architecture implements the create operation using a ‘pull’ mechanism which generates binds between the location in the parent thread’s resources and the child threads’ resources. Consequently, between family creation (i.e. create) and family completion (i.e. sync), resources used for thread function arguments must neither be reassigned nor be reallocated. Moreover, the create operator implements the semantics for multiple control flows. The compiler is aware of control flows modifications at family creation. The control flow separates at the create operator in two parts: the code section right after the create operator and the child thread’s code about to be created. Multiple concurrent control flows generate inter-context dependencies and are captured with SVP inter-thread communication channels. The control flows merge at family completion. Locks on resources (used for thread functions arguments) are being released at family completion. The create operator targets a thread function whose definition is controlled in a similar way to the the sequential paradigm using caller-callee controls following calling conventions.

Other SVP actions have their back-end operators such as kill, break, sync, index, etc. The advantage for the compiler is the visibility of their actions and the verification of their semantics.
4.4 Conclusion

To summarize this chapter, we have looked at the compilation basics for conventional compilers to present the complex and deep process of compiling conventional programs. We then have expressed the differences with SVP compilation in sufficient details to reflect the depth of change necessary to extend a conventional (imperative-language) compiler. This corresponds to the specifications defined for the SVP compiler we have built with an abstract perspective of how it should transform μTC applications. For that purpose, we have given a concise overview of classic compilation to understand the peculiarities of SVP compilation; we have studied the steps involved in the process of compiling a concurrent language in a conventional compilation.

Conventional optimizations are dangerous for any concurrency-oriented program defined with the SVP paradigm and also for concurrency in general. Extreme care has to be taken in that sense for any concurrency-related work in compiler research. The stress on compilation is put on the two major principles of compilations: preservation of code semantics and optimization of the input program in some way. Embedding SVP concurrency, natively within the compilation per se, allows the reuse of the compilation infrastructure, with consequent changes. Consequently, this has a cost on our research. Chapter 5 investigates the potential dangers of introducing concurrency in compiler optimization passes. Later Chapter 6 discusses the research costs for SVP compilation.