On the compilation of a parallel language targeting the self-adaptive virtual processor
Bernard, T.A.M.
Chapter 5

On the challenges of optimizations

A danger foreseen is half avoided

Proverb

This chapter discusses the challenges of conventional compiler optimizations over SVP programs. We have previously discussed in Chapter 3 the properties of the SVP execution model and its implementations. SVP compilation is dissimilar to conventional (imperative) compilation as we have explored in Chapter 4. Some SVP properties clash with the assumptions in optimization algorithms used in modern compilation for imperative sequential algorithms.

Research interests have been put in the dangers of conventional optimizations over concurrency-oriented code. E. Lee [50] simply pledges against thread-based programming models because of new bug appearances in the code using concurrent programming, data-parallel language extensions and message-passing libraries such as PVM and MPI. Gray areas appear for compilers in programs’ description resulting in non-determinism. Boehm [48] explicitly presents problems to compile (with optimizations) concurrent code with Pthreads, a concurrent programming standard. Major problems reside in how concurrency is exposed to the compiler. Sarkar [75] discusses the necessary changes for compiler frameworks to cope with the new paradigm shift of multicore programming. He mentions the indispensable work on compiler optimizations while addressing the challenges of code optimization of parallel programs. Sarkar discusses ongoing evolutionary effort (understand here extending existing methods) to support optimization of parallel programs in the context of existing compiler frameworks, and their inherent limitations for the long term. He then outlines what a revolutionary approach will entail, and identify where its un-
underlying paradigm shifts are likely to lie. In the context of this thesis work, it becomes relevant to investigate the scope of hazards that SVP programs encounter through compiler optimizations. This section sorts out the optimizations that must be constrained to work in this context; but also, it figures which optimizations are not required at all when generating valid Microgrid code; finally, it presents novel optimizations that could benefit the code.

5.1 Hazards with optimizations

This section covers the dangers of conventional optimizations when compiling SVP programs. In Section 4.1.3, the first important principle of compilation is about preserving semantics of an input program. This is our main concern after integrating new compilation schemes into the compiler’s internals. The scientific question here is to observe and to analyze the hazards of program optimizations over concurrency-oriented programs to achieve the second principle of compilation (i.e. optimize the program in some measurable way). In our research implementation, we have decided to embed concurrency constructs into the compiler infrastructure. Therefore, the compiler has an eye on a program’s definition. The validity of internal representations, such as SSA, CFG, DFG, AST, etc., is an immediate concern in providing any compiler optimization the proper abstraction of the problem to transform and optimize. With SVP, the concurrency tree is captured at ‘thread function’ atomicity and the extension of existing IRs with regard to the SVP concurrency constructs.

Purpose of compiler optimizations

During compilation, optimizations can be used a single time or multiple times until an optimal solution is reached. The term optimization signifies that the compiler discovers an optimal solution to a problem against the one taken as input. The gain may be a few operations, a lower use of resources, etc. In the end, the program has to run as fast as the non-optimized one but if possible with fewer instructions, fewer architectural registers, etc. Adding new compilation schemes (cf. Section 4.2) implies the addition of new assumptions in optimization algorithms (typically multiple choices in a switch-case scheme); therefore, there are more chances to break code consistency. Consequently, we have extended the scope of existing optimizations to support the SVP concurrent paradigm.

SVP compilation and concurrent programming

The SVP compilation schemes require proper transformation rules in converting from µTC code (with SVP and C constructs) into the appropriate operations.
On one hand, we discuss here optimizations that can endanger the code validity. On the other hand, optimizations that can be done on the code to make it more efficient, i.e. elimination of non-executable sections, concurrency tree restructuring. In concurrent programming and with SVP compilation, a major issue with concurrent regions is the presence of multiple control flows at the same time, whereas with the sequential paradigm one single flow is present at any time. This change of assumption breaks the sequential paradigm used in some optimizations, for instance, with combining instructions. Some optimizations, individually, are harmless for concurrent-oriented programs. Nonetheless, the combination of optimizations might endanger dramatically the concurrent code without any notification to developers. These unrelated optimizations can have disastrous impact on the code.

5.2 Investigating some optimizations

We have observed, once the code has been flattened into 3-address code, some inter- and intra-procedural optimizations at the middle-end level can potentially break SVP assumptions. Moreover, peephole optimizations (such as combining instructions) are also dangerous, especially for the sake of the synchronized communication channels of the SVP paradigm. Figure 5.1 depicts an example of an unmodified optimization algorithm. The side-effects of this algorithm simply destroy SVP semantics by removing SVP actions from the program’s code.

```
... ...
create(fid;...) foo(x,y); <statement removed>
... ...
sync(fid); <statement removed>
... ...
```

**Figure 5.1:** Example (in the left-hand side) of optimization side-effects on SVP code. In the right-hand side, the dead-code elimination simply removes SVP actions which are considered useless statements, results of an SVP-unaware algorithm. This optimization must be constrained to work in this context.

The SVP compiler is extended with new compilation schemes on top of the existing C-language compilation schemes. The addition of new SVP operations and their corresponding properties requires special attention when interacting with compiler optimizations. The compiler must be aware of what it has to achieve. Moreover, the SVP synchronized communication channels have special semantics that clash with these optimizations, in particular the synchronization events linked while accessing these channels. For instance, a shared communication channel has two ends: an incoming shared (i.e a read from it) and an outgoing shared (i.e. a write to it). A synchronization event is bound on the first read on the incoming shared variable; another synchronization event
is bound on the first write to a outgoing shared variable. Introducing temporary variables, that break down a statement involving a shared variable, can potentially corrupt SVP code consistency. Figure 5.2 illustrates a case where a synchronized communication channel is broken with the introduction of temporary variables.

Figure 5.2: Example (in the left-hand side) of optimization side-effects on SVP communication channels. In the code, ‘x’ is a shared communication channel and ‘a’ a global channel. During the middle-end stage, the synchronization events may be broken with the introduction of temporary variables (‘x.25’ and ‘D.1199’); new reads and writes appear in the program. Note that the local declaration of ‘x.25’ must be propagated to a shared communication channel to keep the SVP semantics valid for subsequent compilation steps.

Compiler optimizations are numerous and we made the following selection based on which optimizations are constrained to work in SVP compilation: SSA, common subexpression elimination, partial redundancy elimination, dead code elimination, combining instruction, copy propagation, instruction reordering.

5.2.1 SSA

The ‘Static Single Assignment’ form, often abbreviated as SSA form or simply SSA, is both a compiler optimization and an intermediate representation (IR) in which every variable is assigned exactly once. Usually occurring at the beginning of the middle-end, the SSA transforms existing variables in the original IR and splits them into versions, new variables typically indicated by the original name with a subscript, so that every definition gets its own version. Consequently, new local variables are introduced in the scope of the function. Doing that allows more aggressive optimizations. Pop [76] presents the benefits of integrating the SSA in an imperative compiler implementation; code restrictions in the SSA form are reduced and diminished hazardous side-effects during optimizations. Figure 5.3 shows the transformation of a piece of code into an SSA representation.

The SSA form utilizes use-def chains which are explicit and each contains a single element (i.e. variable). A use is when the variable is consumed by a read; the def, or define, tags a variable in which a result of an operation is written. For this reason, the SSA form tends to add temporary variables in the local scope of the procedure. When integrating the SVP assumptions into
Chapter 5. On the challenges of optimizations

\[
\begin{align*}
x &= y; & x_1 &= y_1; \\
y &= y \times 10; & y_2 &= y_1 \times 10; \\
x &= y; & x_2 &= y_2;
\end{align*}
\]

Figure 5.3: SSA transformation from a simple source code. In the left-hand side, a piece of source code is the input of the SSA transformer. The result is the SSA form on the right-hand side. The SSA transformation simplifies the properties of variables with their versioning.

the compiler’s internals, the problem arises with SVP communication channels. The semantics of these channels can be broken when transformation into SSA with the introduction of new versioned variables. The synchronization events may simply be corrupted by reads from and writes to these new variables. The idea to solve this problem is to use an ‘evolutionary’ approach as Sarkar describes. In an ‘evolutionary’ manner, compilers incorporate new code in optimization to extend its scope. ‘Revolutionary’ changes are necessary when new paradigm shifts arise and new components are needed and/or compilers need to be rewritten from scratch to support new paradigms. The changes for extending the SSA are to propagate information related to synchronized communication channels into the new versions of variables and handling properly the appearance of temporary variables when these are synchronized communication channels. Therefore, the IR remains correct and safe for coming compiler optimizations.

5.2.2 Dead-code elimination

‘Dead-Code Elimination’ (DCE) is a compiler optimization that removes code that does not affect the program; this useless code is called dead code. Dead code includes code that can never be executed in the program’s code and is often referred to as unreachable code. Moreover, dead code also affects dead variables which become irrelevant to the program. The DCE is a very important and useful optimization, because removing such code has two benefits: it shrinks program size, and it lets the running program avoid executing irrelevant operations, which reduces its running time. The DCE may occur in the middle-end and also in the back-end. As soon as an optimization pass transformed the program internal representation, new dead code may have appeared. Consequently, the DCE is invoked several times during compilation; it basically uses the CFG of the program (cf. Cooper’s and Torczon’s book, pages 498 to 505) and also information from the SSA as Cytron et al. discuss. As illustrated with Figure 5.4, the leaves of the CFG may be useless code when they cannot be reached or simply because they do not return anything while exiting the program’s procedure.

Figure 5.5 illustrates a case where µTC semantics are valid; but, the DCE al-
thread void foo (int x, int y) 
{
    return x + y;

    int z = x * y; /* unreachable statement */
}

Figure 5.4: The DCE algorithm discovers, in this C-language example, that the statement "int z = x * y;" will never be reached since the function exits at the return statement. Therefore, this statement will be removed.

The algorithm has not been yet extended to support SVP assumptions. The statement "x = x", containing a shared synchronized communication channel, is simply removed; even if the statement is valid in \( \mu TC \) and has synchronization semantics. The incoming shared is read and the outgoing shared is written by it. In this case, the DCE algorithm optimizes away the statement.

thread void bar (shared int x) 
{
    ... 
    x = x; 
    ... 
}

Figure 5.5: The DCE algorithm considers that the statement "x = x" is a useless statement with a read and a write to the same variable. However, this is a valid \( \mu TC \) statement with synchronization semantics. This works as a synchronizer on the previous adjacent thread using the shared synchronized communication channel semantics.

Moreover, DCE also discovers useless code which is a piece of code considered not necessary for the computations. Beyond extending the CFG and the SSA forms to support the SVP assumptions, our aim has been to define specific cases in the DCE algorithm where the new SVP operators (such as the create construct or the sync construct) must not be removed and where synchronized communication channels must also not be removed. We follow an ‘evolutionary’ approach to improve the DCE algorithm. Furthermore, the position of an outgoing shared channel (i.e. write into a shared variable) is seen as a useless statement by the conventional DCE algorithm. As a result, this statement will be removed and the SVP program’s semantics are broken. At execution time, the consequence is disastrous; the SVP hardware will encounter a deadlock at this point of the program where the channel awaits something to be written by something that will never occur. Figure 5.6 represents a CFG of thread function “foo” of Figure 6.3.
5.2.3 Common subexpression elimination

‘Common Subexpression Elimination’ (CSE) is an optimization that searches for instances of identical expressions; these expressions all evaluate to the same value. The CSE analyzes whether it is worthwhile replacing them with a single variable holding the computed value. Again, the problem with SVP integration in the compiler infrastructure comes with the synchronized communication channels. Figure 5.7 depicts a case where an expression is substituted. In the case that an substituted expression contains a synchronization channel, the loss of synchronization events simply breaks the SVP programming language’s assumptions. The program’s semantics are therefore broken.

To prevent elimination of statements or substitution of expressions, the CSE algorithm has new cases to handle the presence of communication channels. It simply does not replace these expressions. The solution might make the program’s assembly slightly bigger, but this is the price to pay to support this in the compiler and preserve semantics. Further information about CSE is available in Muchnick’s book [73], pages 378 to 396.
Chapter 5. On the challenges of optimizations

x = y * z + A;
tmp = y * z;
w = y * z * B;
x = tmp + A;
w = tmp * B;

Figure 5.7: CSE works over this piece (in the left-hand side) and replaces the identical expression “y*z” found in statements. In the right-hand side, statements are simpler and smaller which will reduce the number of operations. Nonetheless, operating on statements, where synchronized communication channels are involved, may simply break synchronization events of these channels.

5.2.4 Partial redundancy elimination

‘Partial Redundancy Elimination’ (PRE) is a compiler optimization similar to CSE. The PRE eliminates expressions that are redundant on some but not necessarily all paths through a program. An expression is called partially redundant if the value computed by the expression is already available on some but not all paths through a program to that expression. Cooper’s and Torczon’s book [74], pages 393 to 404, presents PRE algorithms. An expression is considered fully redundant if the value computed by the expression is available on all paths through the program to that expression. This is combined with Lazy Code Motion (LCM) which can move earlier in the program’s path the redundant expression; the removal of a partially redundant expression may reduce the code size of a code section. With SVP, the same case happens as with the CSE and the synchronized communication channels that can break if the PRE algorithm tackles an expression where a channel is involved. Figure 5.8 illustrates an example of PRE over a block of statements.

x = 2 * z * y;
k = 3 * z * y;
w = 2 * z - y;
tmp_0 = 2 * z;
x = tmp_0 * y;
k = 3 * z * y;
w = tmp_0 - y;

Figure 5.8: PRE works over the piece of code in the left-hand side. The expression “2 * z” is partially redundant in the first and third statement. The PRE algorithm moves using LCM this expression at the beginning of the statement list into a new assignment into a new temporary variable. Then, the partial redundant expression is substituted by the temporary variable. Fewer operations will then be produced with the resulting optimized code.

Moreover, PRE can eliminate partially redundant expressions by inserting the partially redundant expression on the paths that do not already compute it, thereby making the partially redundant expression fully redundant. This optimization is important to reduce programmer’s design errors when inserting the same expressions at different code locations. The PRE algorithm is extended
to support the SVP communication channels with care. Insertions and movements of expressions with communication channels are restricted to preserve code consistency.

5.2.5 Combining instruction

‘Combining instruction’, sometimes called combine, is a compiler optimization which occurs at both middle-end and back-end. At both levels, the concepts are similar but there are different assumptions on the operations to combine. At the middle-end level, it ‘combines’ several abstract operations into fewer operations or a single one. At the back-end level, the same happens but with machine instructions instead of abstract operations. The SVP synchronized communication channels are broken by this optimization, where a statement may simply disappear or be merged into another as illustrated in Figure 5.9. The fusion of an incoming shared, or outgoing shared, removes the semantics related to synchronization events.

\[
\begin{align*}
y &= x + 1; \\
z &= y + 1; \\
z &= x + 2;
\end{align*}
\]

Figure 5.9: Example of combining instruction. In the left-hand side, 2 statements with a dependency on variable ‘y’ between them. The combine algorithm discovers a canonical solution of these 2 operations into 1 in the right-hand side. The variable ‘y’ is consequently removed from the resulting code and there are less operations in the optimized code.

The main concern is to protect statements using synchronized communication channels to be modified beyond SVP semantics. To achieve that, an ‘evolutionary’ solution is taken; the algorithm is simply extended to prevent undefined modifications of these statements.

5.2.6 Copy propagation

‘Copy propagation’, sometimes called copy prop, is a generic compiler optimization in which occurrences of targets of direct assignments are substituted with their values. For instance, a direct assignment is an instruction of the form “\( x = y \)”, that simply assigns the value of ‘\( y \)’ to ‘\( x \)’. Figure 5.10 presents a simple example.

Copy propagation works with use-def and def-use chains which provide occurrences of targets. A use-def chain contains the use of a variable and all definitions, all the code locations where this variable is being assigned. In contrast to a def-use chain, it contains the definition of a variable, for instance an assignment in the code, and all the code locations where this variable is used (i.e. read). The compiler takes care of updating these chains during optimizations.
Figure 5.10: Example of copy propagation. The input of the transformation is in the left-hand side. The algorithm of copy propagation yields the code in the right-hand side. The substitution of ‘x’ by ‘y’ reduces the amount of generated operations; the code size becomes smaller.

Further information about the algorithm is available in Muchnick’s book [73], pages 356 to 362. During compilation, the problem remains on the safety of the communication channels while the copy propagation pass occurs.

5.2.7 Instruction reordering

‘Instruction reordering’ is a compiler optimization mainly used with instruction scheduling during code generation. It reorders the instructions of a procedure depending on time-and-cost heuristics. Happening in the back-end while scheduling the program’s instructions, this optimization shuffles earlier or later instructions in the sequence of instructions. For instance, a long-latency operation, such as a load from memory, has more chances to appear sooner in the code to allow for time to obtain its result before it is needed. In other words, the waiting time is interleaved with other instructions. A simple case is illustrated in Figure 5.11 with a create operation and its corresponding sync. The couple of a create operation and its corresponding sync operation must occur first with the create and then its related sync. The semantics become different when the sync operator is pushed by the instruction ordering before its corresponding create; the sync operation will be a no-op, but the create operation will not have its synchronization barrier.

Figure 5.11: Instruction reordering example with the input in the left-hand side (it shows a couple create-sync). A create operation appears first in the sequence of instructions before its corresponding sync. The right-hand side shows a valid executable code, generated by a non-updated algorithm, but with different semantics. The sync operation is then interpreted a no-op and the create operation is not synchronized anymore.

With the SVP operators, the sequence of instructions in the family creation process has to follow a specific order. Another case of instruction reordering is illustrated in Figure 5.12, the create operator is pushed up before the related setting instructions and allocate instruction which allocates the data structure.
for the to-be-created family. The code semantics for the family creation are therefore broken.

\begin{verbatim}
... allocate L3,4,10,7,17 setstart L3,L2 setlimit L3,L4 setblock L3,2 load L14,foo(L1) create L3,0(L14) ...
... load L14,foo(L1)
create L3,0(L14)
setstart L3,L2
setlimit L3,L4
setblock L3,2 allocate L3,4,10,7,17...
\end{verbatim}

\textbf{Figure 5.12:} Instruction reordering example with the input in the left-hand side (it shows a part of the family creation process). The right-hand side shows the result if the instruction reordering algorithm is not correctly extended. In this example, the create instruction appears in the sequence earlier than the allocate which sets up the family entry and the place of computation. At execution-time, the hardware stalls when generating thread family before allocating its data structure. The code semantics are therefore broken.

Therefore, integrating new operators in the back-end must be done with care; the back-end must be aware of the proper sequence of operators before emitting instructions. Introducing a dependency chain throughout the series of instructions solves this problem as shown in Figure 5.13. This optimization is used when the aggressive optimization mode of a compiler is enabled (e.g. optimization flags -O1, -O2 and -O3 in GCC). The dependency chain between operations prevents reordering by the compiler. With respect to SVP communication channels, the same must be done when the scheduler wants to reorder a write to a shared communication before the same channel has been consumed by a read. The extension of this compiler optimization has been done for optimization flag -O1 with an ‘evolutionary’ approach. Optimization flags -O2 and -O3 use their own pass of instruction reordering with a more complex algorithm. In the time frame of this research, these optimization flags have not been investigated; therefore, their use may cause what Figure 5.12 shows.
Figure 5.13: The dependency chain is between blocks and operations. Three distinct basic blocks show the create process: the first for parameter passing (i.e. block A), the second to set up family parameters (i.e. block B), the third to generate the create instruction (i.e. block C). The dependency chain is between this three blocks and prevents the compiler from shuffling them around. Block C depends on block B which depends on block A. The second dependency chain is internal to each block and in the sequence of instructions for preventing what Figure 5.12 illustrates.
Chapter 5. On the challenges of optimizations

5.3 Discussion and conclusion

5.3.1 SVP optimizations

With SVP, in theory, there is no need to have SVP-specific optimizations to already achieve good performance over the Microgrid architecture configurations, as presented in Section 7.2. The Microgrid architecture tolerates high latency instructions. The pipeline interleaves threads’ instructions when one instruction requires data which is not yet ready. Consequently, during this research, investigating SVP-specific optimizations was not a prior interest. However, with SVP compilation, the concurrency tree is the way of representing a program's concurrency. The concurrency tree contains information on each thread family: the maximum number of threads per core (i.e. block) and the ideal place to run threads (i.e. place). Reshaping the concurrency tree of a program by changing the family block size and other family parameters is a potential direction. In order to achieve this, the target must be able to report back to the compiler, in some way, run-time metrics of the selected branch or the entire tree. This is called feedback compilation where a compiler and a target work together to find an optimal way to optimize a program. At run-time, the Microgrid target profiles the behavior of the program’s concurrency tree. Across several executions, the compiler receives relevant information from the target to tune the family and thread information contained in the program’s concurrency tree. The optimizing algorithm will then reshape the workload of thread families and find the best balance over the program’s definition. Moreover, target’s resources are static at design-time; but, they may change at execution-time. The compiler can introduce variables and code sections at compile-time to permit other execution paths. These code sections and variables will be interpreted by the run-time system which sets and optimizes based on this special information. These ideas arose when investigating SVP compilation during this thesis work. The amount of work to implement them was too high considering the priority to get a working compiler to run experiments and the remaining time; therefore, this will be left for future work.

5.3.2 Conclusion

The investigation reported in this chapter contributes to the rest of this thesis where compiler optimizations over concurrent code have to be taken into consideration. This investigation sorted out optimizations that must be constrained to work in this context such as SSA, DCE, CSE, etc. Some optimizations are just not required to gain good performance with the Microgrid configurations such as reordering long-latency operations sooner in the pipeline (the architecture already takes care of doing that at run-time by interleaving threads’ instructions in the pipeline). Some optimizations are novel optimizations that could benefit the code require a so-called ‘revolutionary’ approach. This approach results in adding an entire new component to the compiler in-
Chapter 5. On the challenges of optimizations

structure versus an ‘evolutionary’ approach consisting in extending existing optimization algorithms with new code sections.

With SVP and its implementations, we believe in exposing concurrency constructs directly to the compiler’s internals; therefore, it can be aware of concurrency interactions during program’s compilation. The main objective for compilers is to avoid gray areas in the code in concurrent programming. In order to achieve that, ‘evolutionary’ methods are possible with special handling cases in optimization algorithms. However, for future work, it would be interesting to investigate ‘revolutionary’ methods, such as the ones mentioned in Section 5.3.1. Furthermore, it would be also relevant to investigate interactions of more aggressive optimizations, for instance optimization flags -O2 and -O3 in GCC. Chapter 6 will show the challenges encountered to achieve the extension of the compiler framework and to enable reuse of existing technology (e.g. compiler optimization and imperative compiler infrastructure). To conclude, we have investigated how sequential assumptions clash with SVP concurrent assumptions. The presence of a working SVP compiler proves that, despite the effort and the challenges, the integration of native concurrency in compiler’s internals is possible.