

Static Data-Flow Analysis of Synchronous Programs

Jens Brandt and Klaus Schneider

Embedded Systems Group

Department of Computer Science, University of Kaiserslautern

<http://es.cs.uni-kl.de>

Abstract

Synchronous programming languages are well-suited for the design of safety-critical real-time embedded systems. However, the compilers and synthesis procedures are challenged by the synchronous programming paradigm and have to solve additional problems like causality and schizophrenia problems. Algorithms to solve these basic compilation problems have already become mature, but code optimization still lacks behind. Often, code optimization is left to the back-end tools like compilers for sequential software or hardware synthesis tools.

In this paper, we develop a static analysis procedure to introduce code optimization techniques to synchronous languages. We develop specialized code optimization procedures that can be applied to all kinds of synchronous languages. Similar to the code optimization techniques used for the compilation of sequential software, our procedures are also based on a static data-flow analysis that is adapted to the synchronous programming model.

1 Introduction

Synchronous languages [15, 3] have proved to be well-suited for the design of safety-critical real-time embedded systems consisting of application-specific hardware and software. There are at least three reasons for this success: First, it is possible to generate both efficient software and hardware from the same synchronous program. Second, it is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis. Third, the formal semantics of these languages allows one to formally prove (1) the correctness of the compilation and (2) the correctness of particular programs with respect to given formal specifications [24, 31, 29].

All these advantages are due to the common paradigm of the synchronous languages, which is the *perfect synchrony*. It postulates that a sequence of statements is conceptually executed in zero time. Consumption of time is explicitly

programmed by special statements which partition the program execution into macro steps. From the programmer's point of view, all macro steps take the same amount of logical time, since all variable values change synchronously at macro steps while the micro steps are executed within the same variable environment. Thus, concurrent threads run in lockstep and automatically synchronize at the end of their macro steps.

The introduction of this logical time scale is not only a convenient programming model, it is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, software generated from synchronous programs can be executed on ordinary micro-controllers without complex operating systems. As another advantage, the introduction of a global logical notion of time is the key to provide a solid formal foundation for the semantics of synchronous languages. Finally, since the synchronous model of computation is shared by synchronous hardware circuits, the translation of synchronous programs to synchronous hardware circuits is straightforward [4, 21].

On the one hand, perfect synchrony simplifies programming, since developers do not have to care about low-level details like timing, synchronization and scheduling. On the other hand, the synchronous paradigm has some consequences that make the compilation of synchronous programs not at all straightforward: In particular, causality analysis [30, 32, 29, 35] and schizophrenia problems [31, 38] challenge compilers. Research over the last two decades has tackled these problems, and various compilers have been developed [5, 23, 28, 31, 11, 10, 13, 13, 14, 20] based on different code generation schemes like automaton based code, (boolean) equation systems and concurrent control-dataflow graphs. While the first papers on compilation of synchronous programs focused on the *correctness* of the compilers considering mainly semantic issues like schizophrenia and causality problems, new research efforts now consider the *efficiency* of the generated code: more and more compilers were designed with a target architecture in mind, which provides a better basis for optimization.

This paper also aims at improving the run-time efficiency of synchronous programs. The code optimization we consider in this paper is based on the identification of *passive code*, i.e. code whose results are not required to determine the final result of the computation. Our contribution consists of the definition and implementation of a static data-flow analysis that determines the passive code in synchronous systems which is the key to various optimizations in the course of the further code generation.

However, we neither have a specific realization (hardware or software) or target architecture in mind, nor do we consider a particular source language. Instead, we aim at optimizing an intermediate code that is based on synchronous guarded actions, a well-known and target-independent intermediate format used by many compilers for synchronous languages. This also gives us the possibility to apply our analysis to several source languages and makes it also independent of the target architecture. We implemented all procedures within the upcoming version of our Averest system¹.

Static data-flow analyses on intermediate code are well-known in classical compiler design [2, 16, 17, 22]. However, these analyses cannot be applied for synchronous programs due to the different underlying model of computation and its definition of equivalence of computations: while transformations for traditional sequential languages need to preserve the final values of the variables, transformations for synchronous programs should ensure that the same outputs are produced in each macro step.

The rest of this paper is structured as follows: Section 2 introduces the starting point of our data-flow analysis: guarded actions, which are used as a common intermediate format for the compilation of synchronous systems. Section 3 presents our data-flow analysis and its practical implementations. After we compared our work with classical work in the area of code optimization in Section 4, we list some preliminary conclusions in Section 5.

2 Synchronous Programs

The data-flow analysis which will be presented in the next section does not consider the source program as starting point. Instead, it is based on *synchronous guarded actions* [7, 12, 18, 19], a well-established intermediate code for the description of synchronous concurrent systems. It is used as a common intermediate format in the compilation of synchronous languages, which will not be explained in the following. The interested reader is referred to a detailed description of the compilation process in [6, 31, 28].

Hence, the starting point of our analysis is a system described by a set of guarded actions of the form $\gamma \Rightarrow \mathcal{A}$. The

Boolean condition γ is called the guard and \mathcal{A} is called the action of the guarded action. In this paper, actions are restricted to the assignments of the source language, i.e. the guarded actions have either the form $\gamma \Rightarrow x = \tau$ (for an immediate assignment) or $\gamma \Rightarrow \text{next}(x) = \tau$ (for a delayed assignment). Both kinds of assignments evaluate the right-hand side expression τ in the current environment/macro step. Immediate assignments $x = \tau$ transfer the obtained value of τ immediately to the left-hand side x , whereas delayed ones $\text{next}(x) = \tau$ transfer this value to the following macro step. Guarded actions are generated for the data and the control flow of the program. The data flow consists of all assignments of the program and determines the values of all declared variables. The control flow consists of actions $\gamma \Rightarrow \text{next}(\ell) = \text{true}$ where γ is a condition that is responsible for moving the control flow at the next point of time to location ℓ .

In addition to the set of guarded actions, there are implicit assignments due to the semantics of the program: The so-called *reaction to absence* defines the value of a variable if no action has determined its value in the current (or previous) macro step (obviously, this is the case iff the guards of all immediate actions in the current step and the guards of all delayed actions in the preceding step are false). In this case, the reaction to absence determines the value of the variable, which depends on the storage mode of the variables which is part of the variable's declaration. There are two storage modes, namely *event* and *memorized*: Event variables are reset to their default values (like wires in hardware circuits), while memorized variables maintain their previous values (like registers in hardware circuits). Hence, the absence reactions of each variable may only depend on constant default values or on the previous values of the same variable. In principle, the absence reactions (that are determined by the storage mode) can also be expressed explicitly as guarded actions. However, for practical reasons (in particular to support modularity of the intermediate code [6]) this information is explicitly stored separately from other guarded actions.

The semantics of the guarded actions is defined as follows: The basis is the synchronous model of computation, which divides the run of the program into a sequence of macro steps. In every macro step, every variable has a unique value. The set of possible values is determined by the type of the variable, and the actual value is determined by the current or previous macro steps that currently modify or previously modified the value of the variable (including the reaction to absence). In each macro step i , the system reads *all inputs* x_1^i, \dots, x_m^i and immediately computes *all outputs* y_1^i, \dots, y_n^i , in principle, by evaluating *all* guarded actions. Typically, a system will also have local variables whose values are also determined by guarded actions in the same way as done for output variables, i.e., either by im-

¹<http://www.averest.org>

mediate assignments enabled in the current macro step, by delayed actions enabled in the previous macro step, or by the reaction to absence if neither immediate actions are enabled in the current macro step nor delayed actions were enabled in the previous macro step.

Hence, in each macro step, the guards of *all* actions are simultaneously checked. If a guard is true, then the action is enabled, and is therefore immediately executed. The immediate updates of the variable environment due to immediate assignments raises the question on the causality of enabled actions: actions must not enable themselves to avoid non-determinism or even loss of reactivity. Causality analyses [30, 32, 29, 35] have been developed to analyze the causality of synchronous programs as an additional phase in the compilation.

The system described by the single guarded action for the event variable x $\{(\neg x \Rightarrow x = \text{true})\}$ is a very simple example, which illustrates the problem. Since the assignment of x is executed at the same point of time when x is checked, this system actually has no consistent behavior. If we consider $\{(x \Rightarrow x = \text{true})\}$, then this program becomes nondeterministic, since it has two consistent behaviors. In general, programs like these examples are not desired and compilers for synchronous languages have to reject them.

Consider the synchronous program given on the left-hand side of Figure 1. A compiler extracts four guarded actions for the data flow and two guarded actions for the control flow from the Quartz program `Parallel`. If the program is currently at location ℓ_1 , the local variables x_1, x_2 are set, and depending on them, the outputs y_1, y_2 are set. The control flow is very simple in this example: location ℓ_1 is initially reached, which is set by the action involving the start signal. The delayed action on ℓ_1 is responsible that the label is continuously reactivated.

Obviously, the guarded actions must be executed according to their dynamic dependencies. They can be illustrated by an *Action Dependency Graph (ADG)*, which is a bipartite graph consisting of vertices representing variables, vertices representing the guarded actions, and edges representing the dependencies between the actions and the variables. The edge $(\gamma \Rightarrow \mathcal{A}, x)$ expresses that x is written by action \mathcal{A} , and the edge $(x, \gamma \Rightarrow \mathcal{A})$ expresses that x occurs in the guard γ or in the right-hand side of action \mathcal{A} . Thus, the graph exactly encodes the restrictions for the execution of the guarded actions of a synchronous system. An action can be only executed if sufficiently many variables occurring in its guard and the right-hand side of its assignment (i. e. its read variables) are known. Similarly, a variable is only known if all actions writing it have been evaluated before.

The ADG for our example program is shown on the right-hand side of Figure 1. It reveals that the actions for

x_1 and y_1 and the actions for x_2 and y_2 must be executed sequentially, while both groups can be executed in parallel.

3 Symbolic Data-Flow Analysis

3.1 Read and Write Dependencies

The basis for our data-flow analysis are the read and write dependencies between the guarded actions. We therefore start with their definition:

Definition 1 (Read and Write Dependencies) *Let $FV(\tau)$ denote the free variables occurring in the expression τ . Then, the following definitions determine the dependencies from actions to variables:*

$$\begin{aligned} \text{grdVars}(\gamma \Rightarrow \mathcal{A}) &:= FV(\gamma) \\ \text{rdVars}(\gamma \Rightarrow x = \tau) &:= FV(\tau) \\ \text{rdVars}(\gamma \Rightarrow \text{next}(x) = \tau) &:= FV(\tau) \\ \text{wrVars}(\gamma \Rightarrow x = \tau) &:= \{x\} \\ \text{wrVars}(\gamma \Rightarrow \text{next}(x) = \tau) &:= \{\text{next}(x)\} \end{aligned}$$

The dependencies from variables to actions are determined as follows:

$$\begin{aligned} \text{grdActs}(x) &:= \{\gamma \Rightarrow \mathcal{A} \mid x \in \text{grdVars}(\gamma \Rightarrow \mathcal{A})\} \\ \text{rdActs}(x) &:= \{\gamma \Rightarrow \mathcal{A} \mid x \in \text{rdVars}(\gamma \Rightarrow \mathcal{A})\} \\ \text{wrActs}(x) &:= \{\gamma \Rightarrow \mathcal{A} \mid x \in \text{wrVars}(\gamma \Rightarrow \mathcal{A})\} \end{aligned}$$

$\text{grdVars}(\gamma \Rightarrow \mathcal{A})$ determines the variable occurring in the guard γ , $\text{rdVars}(\mathcal{G})$ determines the variables occurring in the right-hand side expression τ of the assignment. $\text{wrVars}(\mathcal{G})$ returns the set of variables that is modified if guarded action \mathcal{G} fires. In the opposite direction, all guarded actions that read variable x in their guards, read variable x in their action, and write variable x in their action can be determined by $\text{grdActs}(x)$, $\text{rdActs}(x)$ and $\text{wrActs}(x)$, respectively.

3.2 Identifying Passive Code

As already outlined in the previous section, a synchronous program cannot be executed like a usual sequential program due to the paradigm of perfect synchrony. According to this paradigm, all currently enabled actions must be simultaneously executed. Clearly, only a subset of the guarded actions of the program is active within a macro step, and the remaining part is passive. Efficiently identifying this part of the program can significantly improve the run-time performance.

Obviously, guarded actions which are not enabled in the current macro step (their guards evaluate to false) do not contribute to the final result. In addition to disabled

```

module Parallel (event !y1, event !y2) {
  event x1, x2;
  loop {
    ℓ1 : pause;
    {x1 = true; if(x2) y2 = true;}
    ||
    {x2 = true; if(x1) y1 = true;}
  }
}

```

$$\left\{ \begin{array}{l}
\ell_1 \Rightarrow x_1 = \text{true} \\
\ell_1 \Rightarrow x_2 = \text{true} \\
\ell_1 \wedge x_1 \Rightarrow y_1 = \text{true} \\
\ell_1 \wedge x_2 \Rightarrow y_2 = \text{true} \\
\text{start} \Rightarrow \ell_1 = \text{true} \\
\ell_1 \Rightarrow \text{next}(\ell_1) = \text{true}
\end{array} \right.$$

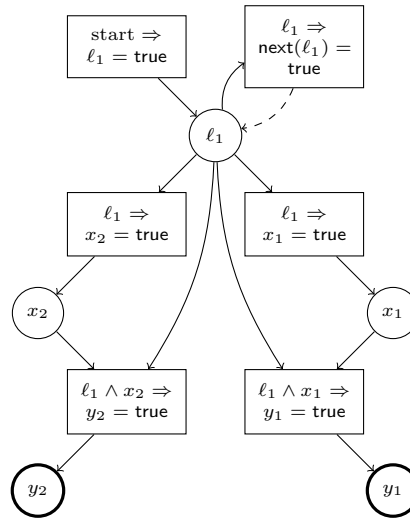


Figure 1. Source Program, its Guarded Actions and the Dependency Graph

guarded actions, there is still more room for optimization: the paradigm of perfect synchrony demands that unique values have to be determined for *all* variables (in particular also for *all* local variables) in each step. However, some of these values may not be needed to compute the final outputs. Although the actions to determine these values may be executed, they do not contribute to the values of the output variables one is interested in. The key to a simple code optimization is therefore to determine actions that do not contribute to the final result, which will be called *passive code* in the following.

The notation of passive code is also known for sequential programs, e.g. for code whose execution is not required to compute the return value of a function. Note that *dead code* is different to passive code, since dead code is not reachable, i.e., not executed in any run of the program. Hence, dead code can be safely removed without modifying the semantics of the program. In contrast, passive code is reachable, and may be required in certain macro steps, but may not contribute to the output values in other macro steps. Hence, passive code must not be removed, but its execution may sometimes be suppressed without modifying the overall behavior of the program. Clearly, we can suppress its execution by an appropriate modification of the guards.

The computation of the situations when code is passive is the problem we want to tackle in this subsection. To this end, we define the predicate req_x for every output and local variable x , which should hold exactly when the value of x is required for the computation of the current macro step.

As all outputs y_i of the system should be computed in all steps, we define $req_{y_i} = \text{true}$ for all output variables². The definition of req_x for local variables x is more complicated and its main idea is to follow the dependencies encoded in the ADG in reverse direction: the predicate is determined for a variable whose successors have been already processed. All actions $\gamma \Rightarrow x = \tau$ which read this variable (as indicated by the outgoing edges) are processed and their partial results are combined in a disjunction: if the variable is in the guard γ , the *required* expression of the target variable req_x is copied. If the variable is in the right-hand side τ only, the *required* expression of the target in conjunction with the guard γ of the action is the desired result, i. e. $\gamma \wedge req_x$.

Unfortunately, this straightforward computation is not possible due to the following two reasons: First, there are data dependencies across macro steps, which result from delayed actions $\gamma \Rightarrow \text{next}(x) = \tau$. Hence, if such a variable x is requested in step i , all variables in γ are required in the preceding step, as well as all variables in τ if γ holds in the preceding step. As an example, consider the variable ℓ_1 in Figure 1. It is set by a delayed action (which is also indicated in the dependency graph by a dashed line), which is always the case for control-flow variables of synchronous programs. Second, the dependency graph is generally cyclic, even for programs which are commonly referred to as acyclic in the synchronous languages community. Again, variable ℓ_1 is an example, since this variable

²These definitions of the *required* predicate of the outputs can be weakened if the system is accompanied by a specification that implies that not all outputs are needed in all steps

occurs on the left-hand side and on the right-hand side of the same action, which leads to a cycle in the dependency graph. For the execution of the synchronous program, this cycle does not cause any problems, since the new value is fed back only in the following macro step, but the definition of the *required* predicate becomes recursive.

In order to determine req_x in the general case, we therefore need a more powerful formalism, which allows us to reason about the cyclic dependencies. It is well-known that most data-flow analyses can be represented as cyclic equation systems whose solution is a least or greatest fixpoint which can be solved by a fixpoint computation similar to model-checking algorithms [34, 36, 33, 1, 27]. In particular, the formal representation as well as the final solution can be done by means of the vector μ -calculus [9, 8, 27]. Provided that the synchronous system has been translated to a Kripke structure [6], any model checker that supports the vector μ -calculus can be used for the computation of the req_x predicates. Furthermore, the underlying theory of vector μ -calculus model checking [27] also answers questions about the feasibility and complexity of the computation.

Hence, we determine the *required* predicates as described in our initial approach above - but with the following additions: If the dependency is across a macro step, we use the diamond operator of the μ -calculus: $\Diamond req_x$ states that there is a successor state in which x is required. To cope with the cyclic dependencies, we use the fixpoint operator $\mu x.f(x)$, which returns the least fixpoint of the given function f . This is exactly the operation that we need in our computation, since we want to minimize the points of time, where req_x for some variable x holds. Thus, the following equation system defines the desired req_x predicates.

Definition 2 (Required Predicate) *The following system of mutually dependent fixpoint equations describes the desired req_x predicates for a set of input/local variables x_0, \dots, x_n and a set of output variables y_0, \dots, y_m :*

$$\left[\begin{array}{l} req_{y_0} = \text{true} \\ \vdots \\ req_{y_m} = \text{true} \\ req_{x_0} \stackrel{\mu}{=} \text{Required}(x_0) \\ \vdots \\ req_{x_n} \stackrel{\mu}{=} \text{Required}(x_n) \end{array} \right]$$

The function Required for the computation of the right-hand sides is defined as follows:

```

function Required( $x_0$ )
 $\varphi := \text{false};$ 
forall  $(\gamma, \mathcal{A}) \in \text{grdActs}(x_0)$ 
  forall  $v \in \text{wrVars}((\gamma, \mathcal{A}))$ 
    case  $v$  of
       $x$       :  $\varphi := \varphi \vee req_x;$ 
       $\text{next}(x)$  :  $\varphi := \varphi \vee \Diamond req_x;$ 
forall  $(\gamma, \mathcal{A}) \in \text{rdActs}(x_0)$ 
  forall  $v \in \text{wrVars}((\gamma, \mathcal{A}))$ 
    case  $v$  of
       $x$       :  $\varphi := \varphi \vee \gamma \wedge req_x;$ 
       $\text{next}(x)$  :  $\varphi := \varphi \vee \gamma \wedge \Diamond req_x;$ 
return  $\varphi;$ 

```

As already mentioned, we can use results from the underlying theory of the μ -calculus for our purposes. These results guarantee that the given equation system has a uniquely determined least fixpoint, since all functions are continuous by construction. The monotonicity can be easily seen in the Required function of Definition 2 since all occurrences of the required-predicates are positive. Continuity immediately follows from this and the finite domain of all variables. The theory also answers the question of computational complexity: the fixpoint can be determined in linear time w.r.t the system size [9, 8, 27].

For the implementation, we used our model checker Beryl, which is part of our Averest system. Actually, apart from the construction of the equation system, no additional effort was needed, since the model checker supports the vector μ -calculus, i. e. it can use the plain-vanilla equation system as an input and return the desired req_x predicates.

To illustrate Definition 2 and the computation, consider the example in Figure 2. The compiler extracts from the source program given in the upper left corner the guarded actions which are shown in the dependency graph on the right-hand side. The equation system is constructed according to the definitions above, which is subsequently solved by the model checker. It determines that y , ℓ_1 , ℓ_2 and ℓ_3 must be always computed, since $req_{\ell_i} = \text{true}$. In contrast, x_1 , x_2 and w must be only computed in every other step, since $req_{x_1} = \ell_1$ and $req_w = req_{x_2} = \text{start} \vee \ell_2$.

3.3 Derived Predicates

The previous section allows us to define the points of time when a variable needs to be read and the points of time when a variable needs to be written. Obviously, a variable should only be read if it is required - and it must be read in order to be required. Hence, the *read* predicate is equivalent to the *required* predicate, and we define $read_x := req_x$.

```

module Interleaved (int ?w, int !y1, int !y2){
  int x1, x2;
  loop {
    y = x2;
    l1 : pause;
    y = x1;
    l2 : pause;
  }
  ||
  loop {
    next(x1) = w * w;
    x2 = w + w;
    l3 : pause;
  }
}

```

$$\left[\begin{array}{l}
req_y \stackrel{\mu}{=} true \\
req_{x_1} \stackrel{\mu}{=} l_1 \wedge req_y \\
req_{x_2} \stackrel{\mu}{=} l_2 \wedge req_y \\
req_{l_1} \stackrel{\mu}{=} \diamond req_{l_2} \vee req_y \\
req_{l_2} \stackrel{\mu}{=} \diamond req_{l_1} \vee req_y \\
req_{l_3} \stackrel{\mu}{=} \diamond req_{l_3} \vee req_{x_1} \vee req_{x_2} \\
req_w \stackrel{\mu}{=} \diamond req_{x_1} \vee req_{x_2}
\end{array} \right]$$

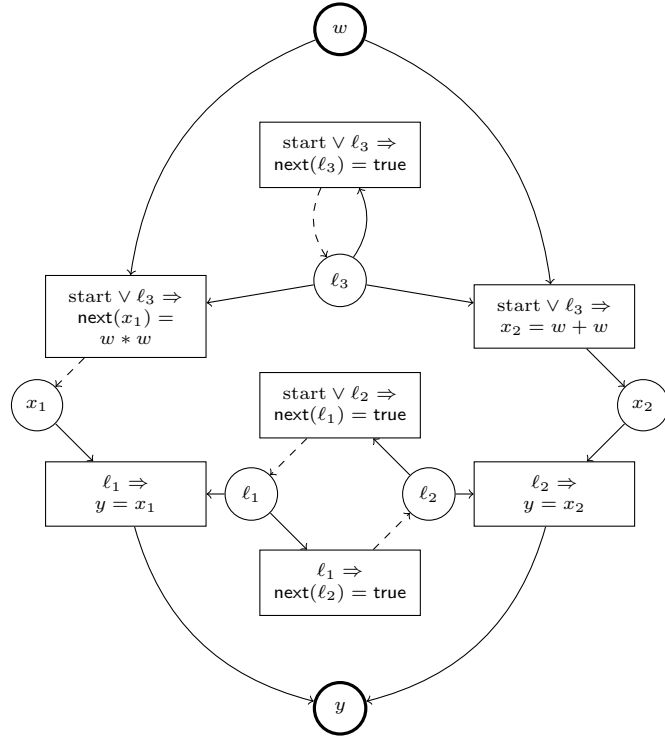


Figure 2. Source Program, Equation System for Required Predicates and Corresponding Dependency Graph

Similarly, a variable is only written if it is required. In addition, there must also be an activated action (or absence reaction) that triggers the write action. To distinguish the different causes for the writing of the variable, we define $init_x$ and $update_x$. A variable is initialized if its scope is entered: Assume that the statement S_x of the source program is the scope of variable x . Then, $init_x$ must be initialized if the statement is started ($go^{surf}(S_x)$), and the variable is required (req_x). Updates are made if one of the actions for variable x writes a value, i.e. if one of the actions is activated $\bigvee_{(\gamma, \mathcal{A}) \in wrActs(x)} \gamma$. All other writes to the variable are made by the absence reaction, which are generally neglectable in the final realizations, since registers store their values without an explicit action. To sum up, we define the following predicates:

$$\begin{aligned}
read_x &:= req_x \\
init_x &:= req_x \wedge go^{surf}(S_x) \\
update_x &:= req_x \wedge \bigvee_{(\gamma, \mathcal{A}) \in wrActs(x)} \gamma \\
write_x &:= init_x \vee update_x
\end{aligned}$$

A well-known definition of classical compiler analysis is the liveness of a variable. A variable is said to be *live* if it has been written somewhere in the past and its value is read strictly before it overwritten again. With the help of our definitions above, the liveness can be simply defined as follows:

$$live_x := \overleftarrow{F} write_x \wedge [read_x \underline{B} write_x]$$

This formula uses temporal logic operators (including past operators) for the definition. The first clause of the definition $\overleftarrow{F} write_x$ ensures that x has been written before (past eventual). The second clause of the definition says that x must be read before it is written again. The strong variant of the *before* operator (as indicated by the underline) guarantees that a read actually occurs.

Again, all the predicates can be computed by a simple fixpoint iteration as done by an arbitrary model checker that supports the vector μ -calculus. In particular, the last definition of $live_x$ can be reduced to an expression of the μ -

calculus, which can be processed without any additional effort [27].

3.4 Conservative Approximations

As it is generally the case for symbolic model checking procedures, the effort to compute the fixpoint of the *required* predicates as described in Section 3.2 can unpredictably vary from system to system. If the computation seems to be too complex for a given system, conservative approximations of the predicates can be applied, which tolerate more reads and writes than necessary. Formally, the condition $req_x \rightarrow \hat{req}_x$ should always hold for such an approximation \hat{req}_x . The same holds for the *live* predicate: the conservative approximation should contain at least the variables that are actually live, i. e. $live_x \rightarrow \hat{live}_x$.

A very rough estimate for all variables is their scope. Especially for local variables which are deeply nested in the program structure and which have a very short lifetime, the following approximation can be very efficient. Apparently, a variable can be only read or written if the control flow is in its scope. Thus, it gives an upper bound for the *required* predicate of the variable.

$$req_x \rightarrow inScope(x)$$

Fortunately, the scope of a variable can be easily determined by

$$inScope(x) := go^{surf}(S_x) \vee in(S_x)$$

using control flow predicates as defined in [25, 28]: the program is in the scope of x if the statement is currently started ($go^{surf}(S_x)$), or if it is already inside ($in(S_x)$). Both predicates are standard predicates for the compilation of the synchronous programs, and they have already been determined if the compilation of the intermediate code as described in [6, 31, 28]. Hence, this approximation does not involve any additional effort.

The *read* and *write* predicates are then approximated similarly: in the worst case, a variable is read and written in each step of its scope, which leads to the approximation $inScope(x)$ for both predicates. Finally, the condition $live_x$ is approximated in the same way, since the worst case is that the variable is set in the first step of the scope and subsequently read in each macro step until the scope is left.

A better approximation also considers the combinational behavior of the system: the approximation that a variable is required in its whole scope is only needed if the variable is able to store its value, i. e. if it is a memorized variable. The *required* predicate can be determined without a fixpoint computation. Definition 2 already returns an acyclic expression, which can be used. Obviously, the previous definitions of $read_x$ and $write_x$ still comply with the approximation.

$$\begin{aligned} \hat{req}_x &:= \text{Required}(x) \text{ if } x \text{ is an event variable} \\ &\quad inScope(x) \text{ otherwise} \\ read_x &:= \hat{req}_x \\ update_x &:= \hat{req}_x \wedge \bigvee_{(\gamma, \mathcal{A}) \in wrActs(x)} \gamma \\ write_x &:= init_x \vee update_x \end{aligned}$$

All correct approximations can be used for a subset of the variables only. Hence, critical parts of the fixpoint computation can be eliminated, while the *required* predicate for remaining variables is still computed.

3.5 Optimizing the Intermediate Code

All the predicates and their approximations which have been described in the previous section can be used in the subsequent code generation step, which transforms the intermediate code to the target code. Our definitions are the basis for the removal of passive code, which improves the run-time of the software code or the energy efficiency of the hardware circuit. Depending on the realization, the information gained by the data-flow analysis can be used as follows:

- The most obvious optimization is to eliminate passive code by using the *required* predicates to strengthen the guarded actions. This is beneficial if software code is generated since only required actions, i.e., non-passive actions, are enabled and therefore executed. As a result, the execution time of the system can be improved. A potential drawback is the additional complexity of the guards due to the evaluation of the required-predicates at runtime: One has to carefully balance the additional effort against the savings.
- This directly leads to the second optimization: If the evaluation of guards is very complex, one can weaken them by allowing the action to fire if a variable is not needed (thus, does not influence the final result).
- If the target is a hardware implementation, strengthening the guards generally improves the energy efficiency of the hardware circuit, since less gates of the circuit operate per cycle. Additionally, the elimination of passive code can be used to optimize the resource requirements of a given system. Generally, a synchronous system needs dedicated resources for all actions that can execute within the same macro step, e.g. if three guarded actions can be potentially run in parallel, and each one of them needs a multiplier, three multipliers must be statically allocated for the hardware realization in principle. If the strengthened guards exclude potential conflicts, the common resource can be safely shared.

- Just as in traditional liveness analysis, the predicate $live_x$ can be exploited to determine an optimal register allocation for the variables. Two variables x_1 and x_2 can be mapped to the same register, if they are never live at the same time, which can be simply checked by $\neg(live_x \wedge live_y)$. Apparently, the formula covers the special case to map two variables of distinct scopes to the same register, as the comments on approximation in the previous section suggest. This optimization is feasible for both hardware and software realizations.

3.6 Example

As an example, consider the Quartz description of a simple pipelined MIPS processor, which is divided into the (classical) five stages instruction fetch (IF), instruction decode (ID), memory access (MEM), execution (EX) and write-back (WB). The Quartz model moreover contains an arithmetic-logical unit and a register file, whereas the instruction/data memory are out of the system boundary so that the memory interface is the observable behavior.

Since all parts of the system are modeled as separate processes running in parallel, they are activated in all macro steps, although some of them might not contribute to the overall result. In particular, if no arithmetic operation is executed, the ALU is nevertheless activated and some output is produced (depending on the implicitly specified operation and arguments), and accesses to the register file are made even if they are not needed for the instruction. These actions are eliminated by adding req_x in all actions for all local variables x . This also includes the special case of a stall, where the whole stage does not contribute to the final result and all its local actions can be safely disabled.

This example also shows us that we can strengthen the initial req_y for the outputs. Obviously, if nothing is read or written to the memory (as indicated by memory read and write signals), neither an address nor a value has to be set at the memory interface. Hence, the initial req_y can be refined in this case.

However, as already pointed out above, strengthening guards may lead to increased performance penalties. Since the data-flow analysis spans over step, it detects that the computation of the next regular PC, which is done in stage IF, will not be needed at the end if a branch is taken instead. The additional condition generally, which prevents the activation, generally duplicates the computations from the other stages (including arithmetic operations and register file accesses). This destroys the pipeline, since its first stage now has the same latency as an unpipelined data-path. In our MIPS pipeline example, we found out that in principle about one third of the actions in the intermediate code can be strengthened. However, merely half of them lead to a performance gain.

4 Related Work

Data-flow analysis has been a static analysis tool for code optimization in classical compiler design for decades. Early work has been presented in [2, 16, 17, 22] and considers different kinds of data flow analyses like the computation of live variables, busy variables, available expressions (to detect shared expressions), used-def chains and many more.

Unfortunately, the data-flow analyses that have been developed so far cannot be directly applied to synchronous languages for the following reasons:

- Classical liveness analysis relies on a sequential control flow, while synchronous languages are inherently parallel. Parallelism is built into the language and is available as an orthogonal construct to all other language constructs. All actions within a macro step are executed in zero-time in parallel. Hence, they are not executed in the order written in the program but according to the data dependencies, which may vary depending on the current inputs of the system (so that dynamic schedules are required for execution).
- The reaction to absence is completely unknown in sequential programming languages, but can be added to the set of guarded actions after linking all separately compiled modules.
- Synchronous programs suffer from schizophrenia problems, which are caused by local variables inside loops. Due to these problems, some instances of variables may be present more than once in a macro step.

Hence, even though static data-flow analysis is a well-established tool for classic compiler optimization phases, it is not yet well developed for the compilers used for synchronous languages. The static analysis described in [37] is used to compute conditions for instantaneous execution of a synchronous program that is done similarly by the control flow predicates in [26]. In contrast to these works, we presented in this paper a more powerful approach that adapts data-flow analysis known from classic compiler optimization techniques.

5 Conclusions

In this paper, we presented a static data-flow analysis for synchronous programs, which can be used for the optimization of the synthesis step. It mainly depends on a definition of a predicate req_x , which describes when a variable x contributes to the final result of the system. We formalized all the predicates with the help of the vector μ -calculus and sketched how the given predicates can be computed with a symbolic state-of-the-art model checker. After showing

some conservative approximations, we finally showed how the computed predicates can be used in the code synthesis step depending on the focused target.

References

- [1] A. Aho, M. Lam, and R. Sethi. *Compilers: Principles, Techniques, & Tools with Access Code: Principles, Techniques, and Tools with Gradience*. Addison Wesley, 2007.
- [2] F. Allen. Control flow analysis. *ACM SIGPLAN Notes*, 5(7):1–19, 1970.
- [3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
- [4] G. Berry. A hardware implementation of pure Esterel. In *Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.
- [5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [6] J. Brandt and K. Schneider. Separate compilation of synchronous programs. In *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM.
- [7] K. Chandy and J. Misra. *Parallel Program Design*. Addison Wesley, Austin, Texas, May 1989.
- [8] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal μ -calculus. In G. von Bochmann and D. Probst, editors, *Computer Aided Verification (CAV)*, volume 663 of *LNCS*, pages 410–422, Montreal, QC, Canada, 1993. Springer.
- [9] R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation-free μ -calculus. In K. Larsen and A. Skou, editors, *Computer Aided Verification (CAV)*, volume 575 of *LNCS*, pages 48–58, Aalborg, Denmark, 1992. Springer.
- [10] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 391–395, Paris, France, 2001. Springer.
- [11] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantics on a sequential execution structure. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5):80–94, 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).
- [12] D. Dill. The Murphi verification system. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, NJ, USA, 1996. Springer.
- [13] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 21(2):169–183, February 2002.
- [14] S. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. In *Synchronous Languages, Applications, and Programming (SLAP)*, Barcelona, Spain, 2004.
- [15] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [16] M. Hecht and J. Ullman. Analysis of a simple algorithm global data flow problems. In *Symposium on Principles of Programming Languages (POPL)*, pages 207–217, Boston, Massachusetts, 1973. ACM.
- [17] M. Hecht and J. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM (JACM)*, 21(3):367–375, July 1974.
- [18] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.
- [19] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.
- [20] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 227–236, Mont Saint-Michel, France, 2003. IEEE Computer Society.
- [21] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In *Proceedings of the Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 195–208, Mook, The Netherlands, 1991. Springer.
- [22] B. Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

- [23] K. Schneider. A verified hardware synthesis for Esterel. In F. Rammig, editor, *Workshop on Distributed and Parallel Embedded Systems (DIPES)*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer.
- [24] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–156, Newcastle upon Tyne, UK, 2001. IEEE Computer Society.
- [25] K. Schneider. *Exploiting Hierarchies in Temporal Logics, Finite Automata, Arithmetics, and μ -Calculus for Efficiently Verifying Reactive Systems*. Habilitation Thesis. University of Karlsruhe, 2001.
- [26] K. Schneider. Improving automata generation for linear temporal logic by considering the automata hierarchy. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2250 of *LNAI*, pages 39–54, Havana, Cuba, 2001. Springer.
- [27] K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.
- [28] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
- [29] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.
- [30] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, DC, USA, 2004. ACM.
- [31] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.
- [32] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.
- [33] T. Schuele and K. Schneider. Abstraction of assembler programs for symbolic worst case execution time analysis. In *Design Automation Conference (DAC)*, pages 107–112, San Diego, CA, USA, 2004. ACM.
- [34] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.
- [35] T. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.
- [36] B. Steffen. Data flow analysis as model checking. In T. Ito and A. Meyer, editors, *Theoretical Aspects of Computer Software (TAPS)*, volume 526 of *LNCS*, pages 346–364, Sendai, Japan, 1991. Springer.
- [37] O. Tardieu and R. de Simone. Instantaneous termination in pure Esterel. In R. Cousot, editor, *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 91–108. Springer, 2003.
- [38] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.