Efficient Code Generation from Synchronous Programs

Klaus Schneider, Jens Brandt, and Eric Vecchié
Reactive Systems Group
Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany

http://rsg.informatik.uni-kl.de

Abstract

We present a new compilation technique for generating efficient code from synchronous programs. The main idea of our approach consists of computing for each program location an instantaneous statement (called a job) that has to be executed whenever the corresponding program location is active. Given the computed jobs, the overall execution scheme is highly flexible, very efficient, but nevertheless very simple: At each instant, it essentially consists of executing the set of active jobs according to their dynamic dependencies. Besides the required outputs, the execution of the jobs additionally yields the set of active threads for the next instant. As our translation directly follows the structure of the source code, the correctness of the translation can be easily checked by theorem provers. Furthermore, our translation scheme offers new potential for multi-processor execution, modular compilation, and multi-language code generation.

1 Introduction

During the last decade, several new programming languages have been discussed in system design like System-C, SystemVerilog, and many others. Among these languages, synchronous languages [2] like Esterel [4] and its variants [14, 21] are particularly interesting for several reasons: First, it is possible to generate both efficient software and hardware from the same synchronous program [3], which allows fast simulation of the application-specific hardware as well as late changes of the hardware-software partitioning of a system. Second, it is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis (since no loops can appear in reaction steps). Third, the formal semantics of these languages allows the application of formal methods to prove (1) the correctness of the compiler and (2) the correctness of a particular program with respect to given formal specifications [19–21, 23].

Although several success stories have already been reported [13] from safety-critical applications like avionic and automotive industries, there is still a need for further research on the efficient compilation of synchronous programs. As observed in [9, 12, 18], fundamentally different techniques have already been developed to compile synchronous programs during the last two decades:

- The first generation of compilers [6] used the *Structural Operational Semantics (SOS)* rules of Esterel to translate the program to an extended finite state machine whose transitions are endowed with corresponding code fragments. As it is well-known, this automaton may have exponentially many states in terms of the size of the given program, which makes this compilation technique applicable only to small programs. Nevertheless, this method generates the fastest code for software on a uniprocessor system, as the compiler generates exactly the code that has to be executed for each particular control state of the program.
- Polynomial compilation was first achieved by a direct translation to *equation systems* [4, 17, 19, 21, 23]. The idea behind this approach is to consider only control flow locations instead of entire control states[1]. The compiler determines (1) the value for every output, and (2) the value of all control flow locations at the next instant of time, both in terms of the current inputs and currently active control flow locations. The approach is particularly efficient for hardware synthesis, since hardware is able to execute all equations in parallel, and the obtained circuits' sizes are only quadratic

[1] We distinguish between control flow *states* and control flow *locations*: A state is a vector of control flow locations, and a control flow location is a place in the program that can hold the control flow for an instant of time. In case of Esterel and Quartz, control flow locations are essentially *pause* statements.

in terms of the given program. Hence, this approach is successfully used for hardware synthesis, and it is still the core of commercial compilers [13]. A software approach on a uniprocessor system, however, has to check all equations one after the other, even those that correspond with control flow locations that are currently not active, which leads to significant performance penalties for programs that contain only a limited degree of concurrency.

- A third approach has been followed by the Saxo-RT compiler [7, 8] of France Telecom, which translates the program into an *event graph*. Hence, an event driven simulation scheme can be used to generate code, which is compiled into efficient C code. In contrast to our approach, this approach does not retain the original program structure and therefore is not able to exploit structural properties of the program like mutual exclusion of substatements in sequences or conditionals.
- A fourth approach is based on the translation of programs into *concurrent control data flow graphs* [9, 10, 12, 18], whose sizes depend linearly on the given program. At each instant, the control flow graph is traversed until nodes are reached that correspond with active control flow locations. Then, the corresponding subtrees are executed which changes the values of the control flow locations (that are maintained in boolean variables).

Our approach is somehow related with the last approach. However, we do not translate the program into control data flow graphs, and neither do we store the control flow in boolean variables. Instead, we generate for each program location a corresponding job that consists of an instantaneous statement (one that does not contain macro steps). Hence, if we know the currently active control flow locations, we simply have to execute the corresponding jobs taking care of mutually dynamic data dependencies in the code. In addition to the computation of the outputs, the execution of the jobs will also generate a set of control flow locations that are active at the next instant of time.

In contrast to control flow graph based approaches, we therefore have no need for a selection tree, i.e. to find in the control flow graph the subtrees that have to be executed next. Instead, we directly execute the jobs that have been directly computed by the previous jobs. Moreover, our compilation technique follows closely the given program structure, so that a formal verification as already done for hardware synthesis [19–21, 23] can be achieved to show the correctness of this approach. This can also be used to obtain a validating compiler that not only delivers compiled code, but additionally a formal proof of the correctness of its compilation. In this paper, we do however not yet consider these topics, and leave them as future work.

The rest of the paper is organized as follows: In the next section, we briefly describe the set of Quartz statements that we consider in this paper. Note that this set of statements is complete in the sense that it allows us to define all other statements as macros, so that our compilation scheme has no particular restrictions. We then define an intermediate language to implement the jobs that are generated by our compiler and describe the translation of Quartz statements to equivalent jobs in detail. After illustrating the compilation by two small examples, we list some preliminary experimental results we obtained with our Averest system[2]. Finally, we conclude with a short summary.

2 The Synchronous Language Quartz

Quartz [19–21] is an imperative synchronous language that shares its basic model with its ancestor Esterel. However, in addition to the many Esterel statements, the language features generic programs (compile time parameters), different forms of concurrency (synchronous, asynchronous, interleaved), explicit nondeterministic choice, arrays, infinite integers and temporal logic specifications. In this paper, we do not rely on these additional features. Therefore, the presented compilation technique can be applied to Esterel programs as well. We consider only the following basic statements:

Definition 1 [Basic Statements of Quartz] *The set of basic statements of Quartz is the smallest set that satisfies the following rules, provided that S, S_1, and S_2 are also basic statements of Quartz, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and α is a type:*

- *nothing (empty statement)*
- *emit x and emit next(x) (boolean event emissions)*
- *$y = \tau$ and next$(y) = \tau$ (assignments)*
- *ℓ : pause (consumption of time)*
- *if (σ) S_1 else S_2 (conditional)*
- *$S_1; S_2$ (sequential composition)*
- *$S_1 \parallel S_2$ (synchronous concurrency)*
- *do S while(σ) (iteration)*
- *[weak] suspend S when [immediate](σ) (suspension)*
- *[weak] abort S when [immediate](σ) (abortion)*
- *$\{\alpha\ y;\ S\}$ (local variable y with type α)*

[2] http://www.averest.org

There are two kinds of (local and output) variables in Quartz, namely *event* and *state variables*, which are manipulated by emission and assignment statements, respectively. State variables y are persistent, i.e., they store their current value until an assignment changes it. Executing a delayed assignment $\mathit{next}(y) = \tau$ means to evaluate τ in the current macro step (environment) and to assign the obtained value to y in the following macro step. Immediate assignments update y in the current macro step and may therefore be rather read as equations than assignments.

Event variables do not store their values, hence, an assignment $y = \tau$ to an event variable just gives y the value τ for this moment of time (unless there is another assignment in the next instant with the same value). If no assignment takes place, the value is not stored, instead, a default value is taken. As most events are of Boolean type, we use the statements *emit* x and *emit next*(x) as macros for $y = \mathsf{true}$ and $\mathit{next}(y) = \mathsf{true}$, respectively.

There is only one basic statement that defines a control flow location, namely the *pause* statement[3]. For this reason, we endow *pause* statements with unique Boolean valued *location variables* ℓ that are true iff the control is currently at location ℓ : *pause*. These variables are directly used as state variables for hardware synthesis, and are used to identify jobs in the approach presented in this paper.

The semantics of the other statements is essentially the same as in Esterel. Due to lack of space, we do not describe their semantics in detail, and refer instead to [19–21] and, in particular, to the Esterel primer [5], which is an excellent introduction to synchronous programming.

3 Computing Jobs from Programs

In this section, we describe the computation of a set of jobs for a given Quartz program. As already outlined, the overall idea of the proposed code generator is as follows: For each control flow location ℓ of the program, a job S_ℓ is computed that has to be executed iff the control flow resumes the execution from location ℓ. Of course, several jobs may have to be executed in one macro step since several locations can be active at once. In addition to the jobs S_ℓ that directly correspond to control flow locations, we additionally use further jobs S_λ to avoid code duplication. As explained below, these additional jobs S_λ correspond to barriers with a lock variable λ.

The overall execution of the jobs is described in more detail in the next section. In this section, we describe first the syntax and semantics of the jobs and their computation.

[3]To be precise, immediate forms of suspend also have this ability.

3.1 The Job Language

In principle, a job S_ℓ consists of a set of guarded actions to implement the data flow of the program and a set of guarded `schedule`(ℓ) statements to implement the control flow of the program. However, we do not compute simple sets of guarded actions and guarded `schedule`(ℓ) statements. Instead, we additionally use further statements like conditional and sequential statements to allow sharing of common conditions. Moreover, we use statements for barrier synchronization to implement the concurrency of synchronous programs. In the following, we discuss these statements in detail.

Definition 2 **[**Job **Language]** *The set of* Job *statements is the smallest set that satisfies the following rules, provided that S, S_1, and S_2 are also* Job *statements, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and λ is a lock variable (integer type):*

- `nothing` *(empty statement)*
- `emit` x *and* `emit next`(x) *(event emissions)*
- $y = \tau$ *and* `next`$(y) = \tau$ *(assignments)*
- `init`(x) *(initialize local variable)*
- `schedule`(ℓ) *(resumption at next reaction)*
- `reset`(λ) *(reset a barrier variable)*
- `join`(λ) *(apply for passing barrier)*
- `barrier`(λ, c) *(declare barrier λ)*
- `if`(σ) S_1 `else` S_2 *(conditional)*
- $S_1; S_2$ *(sequential composition)*

Note that there is no longer a parallel statement and also the abort/suspend statements are no longer required. Moreover, there are no loops, since we can implement them by the help of schedule statements (explained below). Moreover, all job statements are instantaneous[4].

The atomic statements `nothing`, `emit` x, `emit next`(x), $y = \tau$, and `next`$(y) = \tau$ have the same meaning as in Quartz programs. The meaning of conditionals and sequences is also the same as in Quartz. The statement `init`(x) replaces a local variable declaration as follows: when executed, it first removes x from the current context as well as pending (delayed) assignments to x, and then gives x the initial default value according to the type of x.

The `schedule`(ℓ) statement corresponds with a 'delayed goto' to a control flow location ℓ of the Quartz

[4]The job language is therefore also a synchronous language that is however not meant to be offered to the programmer. Instead, it is used as an intermediate language [15] that could, in principle, be the target for many synchronous languages.

program. When executed, it simply puts the label ℓ into a schedule, so that the runtime environment will execute the corresponding job S_ℓ in the next reaction step. Note, however, that $\texttt{schedule}(\ell)$ is instantaneous, so that $\texttt{schedule}(\ell_1); \texttt{schedule}(\ell_2)$ will put both ℓ_1 and ℓ_2 at once into the schedule for the next reaction step.

The statements $\texttt{reset}(\lambda)$, $\texttt{join}(\lambda)$, and $\texttt{barrier}(\lambda, c)$ are used to implement concurrency based on *barrier synchronization*. The statement $\texttt{barrier}(\lambda, c)$ declares a barrier with an integer lock variable λ and an integer constant c as threshold. Executing this statement checks whether $\lambda \geq c$ holds, and if so, it immediately terminates, so that a further statement S can be executed in a sequence $\texttt{barrier}(\lambda, c); S$. If $\lambda < c$ holds, the execution stops, so that this control thread terminates.

Executing $\texttt{reset}(\lambda)$ simply resets $\lambda = 0$, and $\texttt{join}(\lambda)$ first increments λ and then executes a job $\mathcal{S}_\lambda$ that is associated with the barrier whose lock variable is λ. Usually (and in our compiled jobs always), it is the case that the code of job $\mathcal{S}_\lambda$ is a sequence $\texttt{barrier}(\lambda, c); S'_\lambda$ with some job statement S'_λ.

Using the statements for barrier synchronization, it is straightforward to execute parallel code on a uniprocessor machine: We associate with each parallel statement a barrier with lock variable λ and threshold $c = 2$ that is reset when the parallel statement is started. When a thread of the parallel statement terminates, it executes a $\texttt{join}(\lambda)$ statement. If both threads have executed their final $\texttt{join}(\lambda)$ statements, the barrier will be passed, so that the code S'_λ following the associated $\texttt{barrier}(\lambda, c)$ statement in the job $\mathcal{S}_\lambda$ associated with the barrier can be executed.

The implementation of the barrier synchronization for other architectures may (and must) be different. Hence, it depends on the platform that is used to execute the program, while our jobs remain architecture-independent. Different implementations for barrier synchronization already exist [1] for hardware, software on multiprocessors, and software on uniprocessors, so that our jobs can be executed for all of these platforms.

3.2 Computing Jobs

The computation of the jobs is given in Figure 1. To compile a statement S, we start the function call $\mathsf{Jobs}(S, \mathit{nothing}, \{\})$, which computes a tuple $(S_\alpha, \mathcal{P}, \mathcal{F})$ with the following meaning:

- S_α is the initial job of S, i.e., that code that is executed when S is initially started (which is often viewed as being started from an invisible boot control flow location ℓ_α).
- $\mathcal{P}$ is a set of pairs (ℓ, S_ℓ) such that S_ℓ is the job that is associated with control flow location ℓ.
- $\mathcal{F}$ is a set of pairs (λ, S_λ), where λ is the lock variable of a barrier and S_λ is of the form $\texttt{barrier}(\lambda, c); S'_\lambda$ with some threshold c (hence, S'_λ is the job that is executed when the barrier is passed).

The execution of the initial call $\mathsf{Jobs}(S, \mathit{nothing}, \{\})$ will produce subsequent calls $\mathsf{Jobs}(S, S_\eta, J)$ with Quartz statements S, S_η with the following meaning: During the function calls, the statement that has to be compiled has been transformed to the equivalent Quartz statement $S; S_\eta$. Moreover, the set J is either $\{\}$ or a singleton set $\{\lambda\}$. In the latter case, we have to immediately execute $\texttt{join}(\lambda)$ to apply for passing the barrier λ as soon as $S; S_\eta$ terminates. If λ is large enough, the barrier can be passed and the job S'_λ associated with the barrier will be immediately executed.

In principle, our compilation procedure performs a symbolic execution of the statement, and each recursive call corresponds with a SOS rule that defines the semantics of Quartz, which allows us to easily verify its correctness. The recursion is made primarily on S, and secondarily on S_η. We consider the possible cases and explain the correctness of the code given in Figure 1:

If $S \equiv \mathit{nothing}$ holds, we check whether $S_\eta \equiv \mathit{nothing}$ holds. If this is not the case, the correctness of the further recursive call is given due to the fact that $\mathit{nothing}; S_\eta$ is equivalent to $S_\eta; \mathit{nothing}$. In case $S_\eta \equiv \mathit{nothing}$ holds, the compilation of $S; S_\eta$ is completed. However, a $\texttt{join}(\lambda)$ statement may have to be executed next, which is checked by inspecting the set J. There are two possibilities: either $J = \{\}$ or $J = \{\lambda\}$ holds for some barrier λ. $J = \{\}$ means that the current reaction is complete, and therefore this execution immediately terminates. $J = \{\lambda\}$ means that the current reaction is not yet complete and has to continue with an attempt to pass the barrier λ to execute the job S_λ which is done by executing $\texttt{join}(\lambda)$.

The same explanations hold if S is an action. The only difference is that we add the action S to the initial job S_α that is returned in this call.

If a ℓ : *pause* statement is reached, then the execution in this macro step terminates. In the next macro step, we have to execute the job that is associated with the location ℓ. Hence, the job ends with executing $\texttt{schedule}(\ell)$. However, recall that we actually compile $S; S_\eta$, and therefore we have to proceed with the compilation of S_η. The thereby obtained initial job S_ℓ is the job that we computed for location ℓ by compiling S_η.

For a conditional with branches S_1 and S_2, we either execute $S_1; S_\eta$ or $S_2; S_\eta$, depending on the value

function Jobs(S, S_η, J)
  case S **of**
    nothing :
      if $S_\eta =$ *nothing* **then**
        if $J = \{\lambda\}$ **then return** $(\texttt{join}(\lambda), \{\}, \{\})$
        else return $(\texttt{nothing}, \{\}, \{\})$ **end**
      else return Jobs$(S_\eta, \mathit{nothing}, J)$
      end
    emit x, *emit next*(x), $y = \tau$, *next*$(y) = \tau$:
      if $S_\eta =$ *nothing* **then**
        if $J = \{\lambda\}$ **then return** $((S; \texttt{join}(\lambda)), \{\}, \{\})$
        else return $(S, \{\}, \{\})$ **end**
      else
        $(S_\alpha, \mathcal{P}, \mathcal{F}) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
        return $((S; S_\alpha), \mathcal{P}, \mathcal{F})$
      end
    ℓ : *pause* :
      $(S_\ell, \mathcal{P}, \mathcal{F}) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
      return $(\texttt{schedule}(\ell), \{(\ell, S_\ell)\} \cup \mathcal{P}, \mathcal{F})$
    if (σ) S_1 *else* S_2 :
      return CondJobs$(\sigma, S_1, S_2, S_\eta, J)$
    $S_1; S_2$:
      return Jobs$(S_1, (S_2; S_\eta), J)$
    $S_1 \parallel S_2$:
      return ParallelJobs(S_1, S_2, S_η, J)
    do S *while* (σ) :
      $\lambda =$ NewVar();
      $(S_1^\alpha, \mathcal{P}_1, \mathcal{F}_1) =$ Jobs$(S, \mathit{nothing}, \{\lambda\})$;
      $(S_2^\alpha, \mathcal{P}_2, \mathcal{F}_2) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
      $S_{\mathsf{reenter}} = \texttt{reset}(\lambda); \texttt{if}(\sigma)\ S_1^\alpha\ \texttt{else}\ S_2^\alpha$;
      $\mathcal{F} = \{(\lambda, (\texttt{barrier}(\lambda, 1); S_{\mathsf{reenter}}))\} \cup \mathcal{F}_1 \cup \mathcal{F}_2$;
      return $(S_1^\alpha, \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{F})$;
    suspend S *when* (σ) :
      return SuspendJobs$(S, \sigma, 0, 0, S_\eta, J)$;
    weak suspend S *when* (σ) :
      return SuspendJobs$(S, \sigma, 1, 0, S_\eta, J)$;
    ℓ : *suspend* S *when immediate* (σ) :
      return SuspendJobs$(\ell, S, \sigma, 0, 1, S_\eta, J)$;
    ℓ : *weak suspend* S *when immediate* (σ) :
      return SuspendJobs$(\ell, S, \sigma, 1, 1, S_\eta, J)$;
    abort S *when* (σ) :
      return AbortJobs$(S, \sigma, 0, 0, S_\eta, J)$;
    weak abort S *when* (σ) :
      return AbortJobs$(S, \sigma, 1, 0, S_\eta, J)$;
    abort S *when immediate* (σ) :
      return AbortJobs$(S, \sigma, 0, 1, S_\eta, J)$;
    weak abort S *when immediate* (σ) :
      return AbortJobs$(S, \sigma, 1, 1, S_\eta, J)$;
    $\{\alpha\ y; S\}$:
      return $\texttt{init}(x)$; Jobs(S, S_η, J);
  end case
end function

function CondJobs$(\sigma, S_1, S_2, S_\eta, J)$
  $\lambda =$ NewVar();
  $(S_1^\alpha, \mathcal{P}_1, \mathcal{F}_1) =$ Jobs$(S_1, \mathit{nothing}, \{\lambda\})$;
  $(S_2^\alpha, \mathcal{P}_2, \mathcal{F}_2) =$ Jobs$(S_2, \mathit{nothing}, \{\lambda\})$;
  $(S_3^\alpha, \mathcal{P}_3, \mathcal{F}_3) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
  $S_\alpha = \texttt{if}(\sigma)\ S_1^\alpha\ \texttt{else}\ S_2^\alpha$;
  $\mathcal{F} = \{(\lambda, (\texttt{barrier}(\lambda, 1); S_3^\alpha))\} \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3$;
  return $(S_\alpha, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{F})$;
end function

function ParallelJobs(S_1, S_2, S_η, J)
  $\lambda =$ NewVar();
  $(S_1^\alpha, \mathcal{P}_1, \mathcal{F}_1) =$ Jobs$(S_1, \mathit{nothing}, \{\lambda\})$;
  $(S_2^\alpha, \mathcal{P}_2, \mathcal{F}_2) =$ Jobs$(S_2, \mathit{nothing}, \{\lambda\})$;
  $(S_3^\alpha, \mathcal{P}_3, \mathcal{F}_3) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
  $S_\alpha = S_1^\alpha; S_2^\alpha$;
  $\mathcal{F} = \{(\lambda, (\texttt{barrier}(\lambda, 2); S_3^\alpha))\} \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3)$;
  return $(S_\alpha, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{F})$;
end function

function AbortJobs$(S, \sigma, wk, \mathit{immed}, S_\eta, J)$
  $\lambda =$ NewVar();
  $(S_1, \mathcal{P}_1, \mathcal{F}_1) =$ Jobs$(S, \mathit{nothing}, \{\lambda\})$;
  $(S_2, \mathcal{P}_2, \mathcal{F}_2) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
  function AbortMap(S)
    $\texttt{if}(\sigma)\ \texttt{join}(\lambda)\ \texttt{else}\ S$
  end function;
  function DoAbort(S)
    if wk **then** MapSchedule(AbortMap, S) **else** S **end**
  end function;
  $\mathcal{P}_1' = \{(\ell, \text{DoAbort}(S_\ell)) \mid (\ell, S_\ell) \in \mathcal{P}_1\}$;
  $\mathcal{F}_1' = \{(\lambda, \text{DoAbort}(S_\lambda)) \mid (\lambda, S_\lambda) \in \mathcal{F}_1\}$;
  if immed **then** $S_1 =$ DoAbort(S_1) **end**;
  $S_1 = \texttt{reset}(\lambda); S_1$;
  return $(S_1, \mathcal{P}_1' \cup \mathcal{P}_2, \{(\lambda, (\texttt{barrier}(\lambda, 1); S_2))\} \cup \mathcal{F}_1' \cup \mathcal{F}_2)$
end function

function SuspendJobs$(\ell_1, S, \sigma, wk, \mathit{immed}, S_\eta, J)$
  $\lambda =$ NewVar();
  $(S_1, \mathcal{P}_1, \mathcal{F}_1) =$ Jobs$(S, \mathit{nothing}, \{\lambda\})$;
  $(S_2, \mathcal{P}_2, \mathcal{F}_2) =$ Jobs$(S_\eta, \mathit{nothing}, J)$;
  function SuspendMap(S)
    $\texttt{if}(\sigma)\ \texttt{nothing;}\ \texttt{else}\ S$
  end function;
  function DoSuspend(ℓ, S_ℓ)
    if wk **then**
      $\texttt{if}(\sigma)\ \texttt{schedule}(\ell)$;
      MapSchedule(SuspendMap, S_ℓ)
    else
      $\texttt{if}(\sigma)\ \texttt{schedule}(\ell);\ \texttt{else}\ S_\ell$;
    end
  end function;
  $\mathcal{P}_1' = \{(\ell, \text{DoSuspend}(\ell, S_\ell)) \mid (\ell, S_\ell) \in \mathcal{P}_1\}$;
  $\mathcal{F}_1' = \{(\lambda, \text{MapSchedule}(\text{SuspendMap}, S_\lambda)) \mid (\lambda, S_\lambda) \in \mathcal{F}_1\}$;
  if immed $S_1 =$ DoSuspend(ℓ_1, S_1) **end**;
  $S_1 = \texttt{reset}(\lambda); S_1$;
  return $(S_1, \mathcal{P}_1' \cup \mathcal{P}_2, \{(\lambda, (\texttt{barrier}(\lambda, 1); S_2))\} \cup \mathcal{F}_1' \cup \mathcal{F}_2)$
end function

Figure 1. Computing Jobs from Programs

of σ. Hence, we compile both S_1 and S_2, and enclose them in an appropriate conditional statement. To avoid code duplication of S_η, we also use a barrier synchronization with a threshold 1 (this is essentially a function call).

The compilation of a sequence $S_1; S_2$ is very simple: since we actually compile $S_1; S_2; S_\eta$, we only shift S_2 to S_η, and recursively call the function. The correctness is easily seen, since the sequence operation is associative.

For a parallel statement $S_1 \parallel S_2$, we introduce a new barrier λ with threshold 2, and compile S_1, S_2 with that barrier. Moreover, we compile S_η separately, which yields the code that we associate with the barrier λ. Note that we can execute the obtained initial jobs S_1^α and S_2^α in a sequence since they are both instantaneous.

The compilation of a loop statement is explained as follows: we introduce a new barrier λ with threshold 1 that is applied for when it has to be checked whether the loop should be re-entered. Using this barrier, we compute $(S_1^\alpha, \mathcal{P}_1, \mathcal{F}_1) = \mathsf{Jobs}(S, \mathit{nothing}, \{\lambda\})$, which means that we first neglect the additional statement S_η as well as the condition σ and simply execute the loop body S. If S terminates, the corresponding thread executes `join`(λ), since we provided the set $\{\lambda\}$ to this call. In the function that we associate with λ, we check whether σ holds. If this is the case, we execute S_1^α, otherwise, we branch into S_η. The latter is achieved by simply invoking the initial job S_2^α that has been computed by compiling S_η.

function MapSchedule(f, S)
  case S **of**
    `schedule`(ℓ) :
      return f(`schedule`(ℓ))
    `if`(σ) S_1 `else` S_2 :
      S_1 = MapSchedule(f, S_1);
      S_2 = MapSchedule(f, S_2);
      return `if`(σ) S_1 `else` S_2
    $S_1; S_2$:
      S_1 = MapSchedule(f, S_1);
      S_2 = MapSchedule(f, S_2);
      return $S_1; S_2$
    otherwise :
      return S
  end case
end function

Figure 2. Auxiliary Function MapSchedule

The compilation of abortion statements first computes the normal execution that is performed when no abortion takes place. Then, as a postprocessing, we have to add the potential abortion behavior. To this end, we replace for each location ℓ inside the abort statement's body the corresponding job S_ℓ appropriately as shown in Figure 1. The function MapSchedule is thereby implemented as shown in Figure 2. As it can be seen, it applies a function f to every substatement `schedule`(ℓ) in the argument statement S. A similar compilation is used for suspension statements as shown in Figure 1.

In a forthcoming publication (see also [21]) we will present a formal semantics of the job language that will then be the specification of the runtime environment as well as the basis for formal reasoning about compiler correctness.

```
module ABRO(event a,b,r,&o) {
  loop
    abort {
      ell_a : await(a); || ell_b : await(b);
      emit o;
      ellr : await(r);
    } when(r);
}
```

Figure 3. The well-known ABRO Program.

4 Illustrating Examples

Figure 4 shows the resulting jobs that are obtained by compilation of the well-known ABRO program used in many tutorials for synchronous programming (Figure 3). As it can be seen, our compiler has constructed the jobs `f_ell_a`, `f_ell_b`, and `f_ell_r` for the control flow locations `ell_a`, `ell_b`, and `ell_r` as well as the initial job `f_start`. Moreover, there is a barrier job `g_lambda3` that will be activated if the threshold variable `_lambda3` has reached the threshold 2. Clearly, `g_lambda3` is responsible for joining the two threads of the parallel statement.

A more challenging example is given in Figure 5. Figure 6 shows the jobs that are obtained by compilation of this module. As it can be seen, our code generator has constructed jobs `f_q1`, `f_q2`, and `f_q3` for the control flow locations `q1`, `q2`, and `q3` as well as the initial job `f__start`. Moreover, there is a barrier job `g_lambda4` to implement the termination of the parallel statement. Note that this example contains a schizophrenic local declaration, which is correctly implemented by our threads due to several executions of the initialization command `init(x)` within one reaction.

```
void f_start() {
  reset(_lambda3);
  schedule(ell_a);
  schedule(ell_b);
}

void f_ell_a() {
  if(r) {
     reset(_lambda3);
     schedule(ell_a);
     schedule(ell_b);
  } else
     if(~a)
        schedule(ell_a);
     else
        join(_lambda3);
}
```

```
void g__lambda3() {
  barrier(_lambda3,2);
  emit o;
  schedule(ellr);
}

void f_ell_b() {
  if(r) {
     reset(_lambda3);
     schedule(ell_a);
     schedule(ell_b);
  } else
     if(~b)
        schedule(ell_b);
     else
        join(_lambda3);
}
```

```
void f_ell_r() {
  if(r) {
     reset(_lambda3);
     schedule(ell_a);
     schedule(ell_b);
  } else
     schedule(ellr);
}
```

Figure 4. Jobs for Module ABRO.

```
void f__start() {
  init(x);
  if(i) {
     next(x) = true;
     schedule(q1);
  } else {
     reset(_lambda4);
     emit a;
     join(_lambda4);
     if(~x) {
        emit b;
        join(_lambda4);
     } else
        schedule(q2);
  }
}
```

```
void f_q1() {
  reset(_lambda4);
  emit a;
  join(_lambda4);
  if(~x) {
     emit b;
     join(_lambda4);
  } else
     schedule(q2);
}

void g__lambda4() {
  barrier(_lambda4,2);
  emit c;
  schedule(q3);
}
```

```
void f_q2() {
  if(~i) {
     init(x);
     reset(_lambda4);
     emit a;
     join(_lambda4);
     if(~x) {
        emit b;
        join(_lambda4);
     } else
        schedule(q2);
  } else
     join(_lambda4);
}
```

```
void f_q3() {
  if(~i) {
     init(x);
     reset(_lambda4);
     emit a;
     join(_lambda4);
     if(~x) {
        emit b;
        join(_lambda4);
     } else
        schedule(q2);
  } else {
     init(x);
     next(x) = true;
     schedule(q1);
  }
}
```

Figure 6. Jobs for Module Schizophrenia (Figure 5).

```
module Schizo(event i,&a,&b,&c) {
  loop
    {bool x;
      if(i) {
        next(x) = true;
        q1:pause;
      }
      abort {
       emit a; || if(not(x)) emit b;
                  else q2:pause;
       emit c;
       q3:pause;
     } when(not(i));
    }
}
```

Figure 5. A Challenging Example with Schizophrenia Problems.

5 Runtime Environment

In the previous sections, we have shown how a given program can be translated to a set of jobs that correspond with the part of a reaction that is caused by the activation of a particular control flow location. To execute the jobs more or less directly, we must provide a runtime environment like the Esterel virtual machine [16]. In this section, we discuss how such runtime environments can be implemented on uniprocessor and multiprocessor systems.

Regardless of the number of processors that are available for the execution of the program, a general problem is to determine a dynamic schedule according to the (potentially cyclic) dependencies of the actions and their trigger conditions. It is well-known that a program is constructive iff for all reachable states and all inputs, there exists a dynamic schedule to execute the program [11]. We will therefore check at compile time by causality analysis that a dynamic schedule exists. A dynamic schedule can then be determined during runtime by techniques explained in [12]: We execute the active threads until an expression has to be evaluated that requires variables that will be modified during the current reaction. This partial evaluation has to be repeated several times, which

performs a fixpoint computation at runtime similar to the fixpoint computation done by causality analysis at compile time [11, 22, 24].

However, our case studies have shown that, in many cases, there is a much more efficient alternative: If the jobs do not have internal cyclic dependencies (note that this does not imply that the threads of the synchronous program do not have cyclic dependencies) and if there is a topological ordering of the jobs' dependency graph, then a static schedule can be computed at compile time. Hence, the jobs of a reaction will be executed in the given static order without a special treatment of immediate actions: They update the environment immediately. Moreover, taking the inputs of a program into account, schedules depending on different input conditions can be precomputed, so that programs with pseudo-cyclic behavior can be handled.

An important special case are programs that do not contain immediate actions: these programs are trivially free of cyclic dependencies and therefore, we can execute all jobs independently of each other in any order.

To establish the correct resumption of the execution between different macro steps, we maintain two global sets: (1) a set of actions $\mathcal{A}_{\mathsf{next}}$ that stores all data actions that are scheduled from the current reaction for the next instant, and (2) a set of locations $\mathcal{L}$ that consists of program locations where the control flow has to be resumed a the next instant of time. In the following, we describe the manipulation of these sets by the jobs.

Clearly, nothing has to be done for the statement `nothing`, and immediate actions have to immediately update the current environment (respecting cyclic dependencies), whereas delayed assignments are simply inserted in the set $\mathcal{A}_{\mathsf{next}}$ after evaluation of their right hand sides.

The execution of local variable initialization, i.e., of statements `init`(x) immediately assigns $x = \tau_0$ and removes all pending actions on x that are scheduled in the set of actions $\mathcal{A}_{\mathsf{next}}$ (since these refer to the old scope that has now been left).

The `schedule`(ℓ) statement simply inserts the location ℓ in the set $\mathcal{L}$ such that the corresponding job will be invoked in the next instant. Note again, that the execution of `schedule`(ℓ) does not take time. Hence, a sequence like `schedule`(ℓ_1); `schedule`(ℓ_2) will insert both ℓ_1 and ℓ_2 in the set $\mathcal{L}$.

The execution of conditionals and sequences is straightforward, provided that we respect cyclic dependencies that are of particular interest for evaluation of the condition in conditional statements. However, there are no additional problems that are worth to be mentioned.

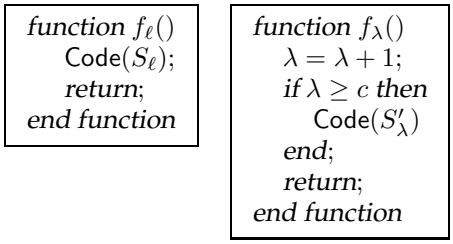

Figure 7. Executing Jobs on a Uniprocessor System.

The implementation of the statements for barrier synchronization, i.e., `reset`(λ), `join`(λ), and `barrier`(λ, c) is more challenging, since it depends on the available platform: On a *uniprocessor system*, we know that there will be no simultaneous access to a variable, thus we can directly implement the barrier synchronization as follows: we use a globally visible integer variable for the lock variable λ. A `join`(λ) will then directly invoke a job S_λ that is implemented as shown in Figure 7. This job first increments the lock variable, and if the threshold is reached, it proceeds with the execution. Otherwise, it immediately terminates and waits for further invocations by other `join`(λ) statements. Clearly, `reset`(λ) resets a lock variable to the initial value 0 (which may be necessary due to re-entering by a loop even in the same reaction).

On a multiprocessor system, the above solution can not be used. As we want to make use of the available processors, we schedule different jobs to different processors. However, it may then be the case that more than one `join`(λ) statement with the same lock variable λ will be executed. There are well-known solutions to implement barrier synchronization on multithreaded systems. A particularly simple solution is to make use of shared counters [1], i.e., each job has its own lock variable which is incremented by execution of a `join`(λ) statement. The `barrier`(λ, c) statement then has to sum up all related counters, which only requires read access, and is therefore safe in a multithreaded environment.

6 Experimental Results

We implemented the job-based compilation for our synchronous language Quartz and also a runtime environment for the obtained jobs on uniprocessor machines. We selected different benchmarks from the Averest benchmark set[5] and generated C code for them

[5] `http://www.averest.org`

with both the new job-based compilation scheme and the traditional equation-based technique [23]. Both versions were compiled with gcc 4.0 and their runtimes were measured on a Pentium 4 (2.8 GHz) machine running a Linux 2.6.12 platform.

Table 1 shows the experimental results, where column 'iter.' lists the number of repeated executions of the benchmark given in the corresponding row (in case of ABRO, we list the number of executed reactions). Column 'size' describes the value of the generic parameter of the benchmark (number of events for ABRO, bitwidth for AddSerial, and the size of the array for BubbleSort).

As it can be seen, the runtime of the job-based compilation scheme is not always better than the runtime of the equation-based code, but in many cases, it outperforms its competitor by far. Having a closer look at the examples, the reason for this behavior becomes clear: The efficiency of the job-based code depends to a high degree on the amount of concurrency available in the Quartz program.

Concurrency can occur in both the control flow and the data flow, i.e. the amount of concurrency is given by the relative number of locations that are active in each step and how many data values must be computed in each step. Since the equation system computes all of them in each step, highly-parallel programs, which require these computations, are perfectly handled by the equation-based code. ABRO perfectly fits in this category: It is inherently parallel so that the equation-based code is optimal. The worse execution time of the job-based code is caused by the overhead of the runtime environment: The scheduling of jobs and the execution of actions must be additionally handled.

However, the equation-based code does not perform well for sequential software, which favours sequences and modifies only a small part of the data values in each macrostep. Since the job-based code only executes the relevant assignments, it is much more efficient in this case. For example, in the bubble-sort algorithm, only two elements are modified in each step. Nevertheless, the equation-based code checks all assignments for execution in each step. Thus, by scaling the array size, the equation-based code of the bubble-sort algorithm can be made arbitrarily inefficient in comparison to the job-based code.

In the serial adder example, there is a high data flow parallelism, but the control flow is almost completely sequential. This increases the number of jobs as well as the number of equations. However, whereas the larger number of program locations is irrelevant to the job-based code, the additional equations require more runtime. Again, by scaling the program size, the equation-based code gets more and more inefficient in relation to the job-based code. However, as it can be clearly seen, the impact of control flow is not as high as the data flow, since the control flow equations tend to be much simpler.

If we compare the sizes of the generated binaries, it can be seen, that the code size obtained by the job-based compilation grows more slowly than the one obtained with the equation-based compilation. However, it requires some initial costs due to the linkage of the runtime environment (6kB) and the bitvector library (6kB) which amortizes for larger programs.

All in all, it can be concluded that both compilation schemes are complementary. For hardware-oriented modules and programs that have a high degree of (data-flow) parallelism, the equation-based code performs better. For software-oriented code, where only a small part of the entire code is active and where only a few variables are modified in each step, the job-based code tends to be more efficient.

7 Summary

In this paper, a new code generation scheme has been presented that is based on computing jobs that correspond with the control flow locations of the given program. We believe that this compilation technique will be useful for many other purposes that we will consider in our future work. We already discussed the runtime environments for various computer architectures that are straightforwardly implemented for the jobs. Moreover, modular compilation [25] is simplified to a large extent, since the job code explicitly contains the surface of the program given as the initial job `f_start`. This is necessary when interactions between pre-compiled modules have to be considered.

References

[1] G. Andrews. *Concurrent Programming – Principles and Practice*. The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[2] 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.

[3] G. Berry. Synchronous languages for hardware and software reactive systems. In C. Delgado Kloos and E. Cerny, editors, *Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, Toledo, Spain, April 1997. Chapman and Hall.

[4] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT, 1998.

[5] G. Berry. The Esterel v5_91 language primer, June 2000.

example	size	iter.	equation-based code		job-based code	
			runtime	size	runtime	size
ABRO	5	10^9	479 s	11 kB	756 s	19 kB
	10	10^9	1048 s	14 kB	1542 s	22 kB
	20	10^9	2267 s	21 kB	3338 s	27 kB
AddSerial	8	10^9	1959 s	17 kB	1330 s	20 kB
	16	10^9	18740 s	29 kB	2834 s	24 kB
BubbleSort	8	10^6	27.5 s	60 kB	9.2 s	18 kB
	16	10^6	274 s	37 kB	21 s	19 kB
	32	10^6	6050 s	192 kB	84 s	19 kB
IslandTrafficController	–	10^9	1477 s	28 kB	908 s	28 kB

Table 1. Benchmark Results

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] 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 *Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 391–395, Paris, France, 2001. Springer.

[8] 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), 2002.

[9] 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.

[10] S. Edwards. ESUIF: An open Esterel compiler. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5), 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[11] S. Edwards. Making cyclic circuits acyclic. In *International Design Automation Conference (DAC)*, pages 159–162, Anaheim, California, USA, 2003. ACM.

[12] S. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. In *Synchronous Languages, Applications, and Programming (SLAP)*, Barcelona, Spain, 2004.

[13] Esterel Technology. Website. http://www.esterel-technologies.com.

[14] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *International Design Automation Conference (DAC)*, pages 511–516, New Orleans, Louisiana, USA, 1999. ACM.

[15] J.-P. Paris, G. Berry, F. Mignard, P. Couronne, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire. Projet SYNCHRONE: les formats communs des langages synchrones. Technical Report 157, Irisa, 1993.

[16] B. Plummer, M. Khajanchi, and S. Edwards. An Esterel virtual machine for embedded systems. *SLAP'06*, 2006.

[17] A. Poigné and L. Holenderski. Boolean automata for implementing pure Esterel. Arbeitspapiere 964, GMD, Sankt Augustin, 1995.

[18] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 227–236, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[19] 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.

[20] 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, June 2001. IEEE Computer Society.

[21] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, 2006.

[22] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington D.C., USA, September 2004. ACM.

[23] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2006.

[24] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, June 2005. IEEE Computer Society.

[25] J. Zeng and S. Edwards. Separate compilation of synchronous modules. In *International Conference on Embedded Software and Systems (ICESS)*, Xian, China, 2005.