Synthesis for VLIW Architectures

Daniel Baudisch

DIPLOMARBEIT

eingereicht am
Fachbereich Informatik der Technischen Universität Kaiserslautern

Mai 2008

Betreuer:

1) Dr. Jens Brandt

2) Prof. Dr. Klaus Schneider

Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mir bei der Erstellung dieser Arbeit geholfen haben. Dazu zählen in erster Linie meine Betreuer Prof. Dr. Klaus Schneider und Dr. Jens Brandt. Des Weiteren danke ich besonders meinen Eltern, die mir immer zur Seite standen und mich immer unterstützt haben. Zudem möchte ich auch den Mitgliedern der Arbeitsgruppe Eingebettete Systeme für ihre Unterstützung bei der Erstellung dieser Arbeit danken.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Kaiserslautern, am 14. Mai 2008

Daniel Baudisch

Abstract

This thesis considers the translation from concurrent guarded actions to sequential code which is optimized for VLIW architectures. Guarded actions are an intermediate format for various synchronous languages, which are convenient for modeling embedded systems. These systems usually consist of at least one processor, which is described using the synchronous language. Synchronous languages provide implicit parallelism of instructions, i.e. they are a natural description of a set of gates, which physically run in parallel. Before creating a prototype of a new developed processor, it is often simulated by translating it into a software model. Therefore, its description has to be sequentialized to simulate the model on a convential sequential processor. Beside the investigation of efficient sequentializing algorithms, another challenge that came with the development of parallel processors was to use the parallelism provided by synchronous languages for a better exploitation of the parallelism of these processors. Beside multi-core processors, one type of these parallel architectures is the VLIW processor, which is capable of executing so-called bundles. A bundle is a list of instructions that have been explicitly prepared by a compiler for parallel execution.
To exploit the capabilites of VLIW architectures and to achieve optimal results, the algorithm presented in this thesis creates a static schedule of the guarded actions. This static schedule contains a list of bundles, whereas each bundle contains instructions for a set of data independent actions that can fire at the same time. For further optimizations, the partial evaluation is applied to the sequentializing algorithm. This technique duplicates the code and fixes the value of a variable to a constant. Hence, this application leads to simplifications, which can reduce the number of guarded actions in the code blocks. In general, the overall size of the code will be increased, but the number of instructions that have to be executed for a specific input are reduced. The application of sequentializing using partial evaluation resulted in a speed-up of the simulation runs, even in small examples.

Contents

1 Introduction

1.1 Motivation

Computer systems are more and more ubiquituous in our everyday lives. Often they appear as embedded systems, e.g. in cars, aircrafts or multimedia systems. These embedded systems are usually application-specific hardware software systems to get the optimal performance at low costs. The disadvantage of creating such a system is that early decisions have to be made about the HW/SW-partition. This problem is often defered by the use of architecture-independent models of the embedded system. The HW/SW partitioning is then done in a later step of the development; in general, after the completion of the system. Hence, parts of a system that was described in a synchronous language needs to be compiled to sequential code.
A standard approach is to generate equational code, which is very slow because a lot of code is executed, even if it is not necessary. For every variable, a new value is calculated, although one or more output variables might keep their values. Many researchers investigated methods and algorithms to improve the creation of sequential code. Many of these algorithms remove the entire concurrency of the program although each modern processor makes use of parallelism. We can observe that the computational requirements of embedded systems grow at steady rate, which leads to the requirement of generating fast code.
Another important point is the development of an embedded system. Although verification is the essential part to show the correctness of a system, testing is still important in the process of developing an embedded system. To be able to run many test cases in a short time the generated code for testing purposes should be as fast as possible. Again, the goal is to generate fast code.
Synchronous languages offer explicit concurrency which seems to be perfect for creating software for multicore processors. A lot of work has been done in creating multithreaded code, but usually the threads that are created of such a synchronous program are very small. The costs for thread synchronization may be higher than the performance boost gained by executing the threads in parallel.
An alternative processor architecture supporting parallelism is the EPIC (**e**xplicitly **p**arallel **i**nstruction **c**omputer) architecture which is also known as VLIW (**v**ery **l**arge **i**nstruction **w**ord) architecture. An advantage of this architecture aside from the ability to execute instructions in parallel is the reduction of power consumption, which is very important for most embedded systems.

1.2 Contribution

This thesis presents a procedure to create sequential code from synchronous programs which is optimized for VLIW architectures. That means that special care is taken of the implicit parallelism of the synchronous language, but also other optimizations will be implemented to gain a speed-up in executing the synthesized program. The main optimization is the so-called partial evaluation, which is intended to reduce the size of code that has to be executed in one cycle. However, the disadvantage of the partial evaluation is the increase of the overall size of the code. Therefore, the partial evaluation is applied in different ways.

1.3 Related Work

Edwards presents in [9] several compiling techniques that are used to compile synchronous languages into sequential code. Halbwachs et al. [12] describe an algorithm for compiling Lustre programs. Lustre prohibits cyclic dependencies, therefore, they always find a schedule to optimize their logic network. Chiodo et al. [7] attempt to reduce the size of the synthesized code by sharing duplicated code. They represent the automaton and its branching programs as a single reduced, ordered BDD [2]. Zeng et al. [32] give an algorithm to produce sequential code from arbitrary acyclic program dependence graphs. However, the guarded actions build dependence graphs which may contain cycles; therefore, it does not meet the requirement of Zeng et al. Cyclic dependencies can be removed by the method presented by Malik [18], however this requires the usage of ternary logic, i. e. more variables, and the usage of iterative calculations. Edwards [10] presents a technique to break up cyclic dependencies for equational systems using BDDs. However, this thesis does not create equational code. Another way to break up cyclic dependencies.is the application of partial evaluation, which is actually intended to create specialized fast code. In [19], Jones et al. give an introduction to partial evaluation. The idea of partial evaluation is that a function with two arguments is transformed to a function with one argument by fixing the other argument. One can say, that a generic solution is split into one or more specialized solutions. This technique can also be applied to code by replacing a variable by a constant value. Furthermore, this has to be applied for each value that can be reached by the variable that has to be replaced. The main disadvantage of partial evaluation is the duplication of code, assuming that Boolean values are applied to the code. However, the application of partial evaluation can reduce a subblock drastically, giving a high speed-up in the execution of the code. In this thesis, the partial evaluation is only applied to Boolean variables yielding some subblocks, whereas the decision which has to be executed is made by a `if-then-else`-statement. A compiler for a VLIW architecture is able to transform this statement depending on the size of its subblocks to two blocks that are reached by conditional branches or by executing both paths using predicated execution [20]. Wall [30] gives an overview of techniques to improve the parallelism in programs. Most of these techniques must be implemented by the compiler that has precise knowledges of the target architecture, e. g. software pipelining, software branch prediction or trace scheduling.
Wenz [31] gives a description of synthesizing synchronous languages. He starts with a detailed description of synchronous languages and the problems that are introduced with them,

i. e. causality and schizophrenia problems In his work, Wenz creates an equational system of the compiled program and outputs it as so-called sequential circuit code. Equational systems are used to create hardware circuits and allow symbolic model checking of the created circuit. However, the disadvantage of this code is its slowness, when it is executed in software.
Another approach of running synchronous code on a processor is given by Li et al. [16], [17]. They developed a processor that is capable of execution Esterel code. The core of the processor is called the KEP3a Reactive Multi-threaded Core and is scalable, i. e. depending on the used FPGA board, multiple cores can be connected to increase the speed of the execution of Esterel code.

1.4 Organization of the Thesis

This diploma thesis is organized as follows: Chapter 2 starts with an introduction to synchronous languages, in particular Section 2.1 introduces to Quartz, which is a representative for synchronous languages. Section 2.2 proceeds with an explanation of guarded actions, which are the input for the synthesis. This is followed by Section 2.4 that gives a short description of VLIW processors. The main part, the description of the implemented algorithm, is split into four parts. It starts with a description of the input data for the synthesis in Section 3. This section also describes the rearranging of the input data for a convenient representation of all necessary data. Section 4 describes the sequentialization process that is applied to the rearranged data. The sequentialization process creates the so-called bundle code, which can be optimized after the sequentializing by the algorithm described in Section 5. The optimized bundle code can be translated into C code as shown in Section 6. Finally, Section 7 gives the results of the taken benchmarks, and draws the conclusions.

2 Preliminaries

2.1 Synchronous Languages

Synchronous languages such as Esterel [5], [1], Quartz [22], [25] or Lustre [11] have been designed to model real-time systems, no matter whether in hardware or in software. One of the key features of these languages is to abstract from time and physical conditions like gate delays [26]. Hence, the programmer is able to keep the focus on implementing the logic of a circuit without being deflected by low-level issues. Another important aim of synchronous languages is to be able to create parallel processes without the additional effort of thread managment, which is necessary in conventional languages like C.

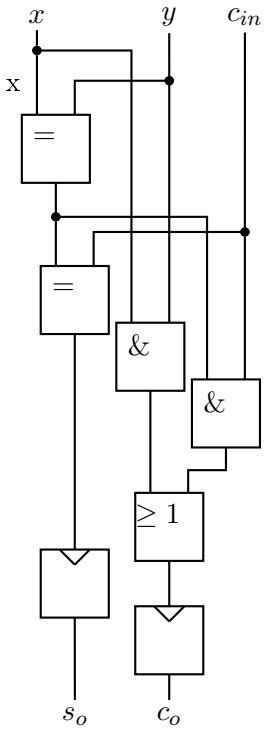
Quartz

Since all programs in this thesis were written in Quartz, this chapter introduces Quartz as a synchronous language. Before implementing any behavior of a system, we have to define an interface to our environment that declares input and output variables. The behavior of a module is defered by a statement. Statements can be divided into two classes, the so-called micro steps and macro steps. The former do not consume time, while the latter consume exactly one logical unit of time. Consuming no time can be thought of executing an instruction in an infinitely short time. Macro steps consist of finitely many micro steps. All micro steps in a macro step are executed in parallel. To get an impression how this can work, we can imagine a circuit where the current flows with an infinite velocity through gates and wires and where it can be only delayed by latches. The gates and lines represent the micro steps while the entire circuit including the latches represents the macro step. Usually macro steps are indicated by a `pause` statement. There are many other statements that consume time but they can all be replaced by micro steps and pause statements.

Figure 2.1 shows a full adder in a pipeline of a processor, i.e. latches are appended to the combinational circuit to store the result. The circuit of the adder is given on the left hand side and its implementation in Quartz is given on the right hand side. Declarations of local identifiers have been removed to focus on the essential code. As can be seen, the gates are represented as logical equations and the latches are programmed with so-called `next`-statements.

Basic Statements

This section presents the basic statements of the Quartz language [22], [24], [23]. The core language of Quartz is powerful enough to define many other statements as simple syntactic sugar and consists of the following basic statements:




```
module FullAdder(bool x, y, cin, &s, &cout) {

   a = x xor y;

   b = a xor c;
   c = x & y;
   d = a & c;

   e = c | d;
   next(s) = b;
   next(cout) = e;
}
```

Figure 2.1: Full Adder Circuit and Implementation in Quartz

Definition 1 *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)*
- *$y = \tau$ and $next(y) = \tau$ (assignments)*
- *ℓ : pause (consumption of time)*
- *$if(\sigma)$ 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)*
- *choose S_1 else S_2 (nondeterministic choice)*
- *assume(φ) (inline assumption)*
- *assert(φ) (inline specification)*
- name($\tau_1, \ldots, \tau_n$) *(module instance)*

The labeled `pause` statement is the only basic statement that consumes time. For this reason, pause statements are endowed with unique Boolean valued location variables *ell* that are true iff the control is currently at location *ell* : pause. These variables are used to define the control flow of the module. Using the other statements unless the assignments, the control flow can be manipulated.
The assignments represent the statements to control the data flow of a module. There are two kinds of assignments. Immediate and delayed assignments. An immediate assignment assigns a variable y a value that is obtained by the evaluation of τ in the same macro step. In contrast, the delayed assignment evaluates τ in the same cycle, too, but assigns the obtained value to y in the next macro step. Furthermore, the language Quartz knows two kinds of variables, namely event and state variables. The former do not store their value. They only keep a value in the macro step where it was assigned. In contrast, state variables are persistent, i. e. they store their current value until an assignment changes it.
As mentioned, the Quartz language consists of additional statements that are syntactic sugar, i. e. they can be rewritten using the statements defined in 1. To give an example consider the statement `ell: await y`, which is also known from the language Esterel. As one can see, the statement is labeled; therefore, it may consume time. The semantics of this statements are to wait for the presence of a variable `y`,i. e. `ell: await y` waits until `y` is set, but at least one cycle. Hence, it can be expressed by some core statements as follows: `do { ell: pause; } while(!y);`. Another example is the `always` statement, which is defined as follows: `always S;` $\equiv$ `do { S; pause; } while(true);`.

Causality

The implicit parallelism of synchronous languages introduces some problems that do not occur in sequential languages. The semantics of synchronous languages allow to write programs that are syntactically but not semantically correct [4].

- Logical correctness: A program is logically correct, if it is reactive and deterministic [5, Chapter 4], [15], [22, Section 3.2.2]. In particular, that means that a program has to react for each input with exact one output. Synchronous composition of control flow and macro steps can complicate the semantic analysis of a program. As mentioned it is possible to write programs that are syntactically but not semantically correct. Consider

```
module P1(bool &o) {
   if(o) o=true;
}
```

Figure 2.2: A non-reactive system.

the Quartz programs in Figure 2.2 and 2.3. Module P1 provides two possible outputs (`o=false` and `o=true` are correct outputs). In contrast, module P2 does not provide any outputs due to the contradictory `if` statement. Note that the assignments and the

```
module P2(bool &o) {
   if(!o) o=true;
}
```

Figure 2.3: A non-deterministic system.

`if` statements are executed in parallel. That means that an assignment in a conditional statement can influence the result of the conditional statement.

- Constructiveness: A synchronous program is constructive, if its output values can be computed without guesses. The constructiveness restriction is necessary to be able to synthesize a module; therefore this restriction is expected for all modules that are intended to be synthesized.
- Write conflicts: Another problem that is introduced by the concurrent execution of instructions are write conflicts [15]. Due to the support of immediate and delayed assignments, there are three types of write conflicts (see Figure 2.4,2.5 and 2.6). All

```
module WRCONFLCT1(int &o) {
   o = 1;
   o = 2;
}
```

Figure 2.4: Write conflict, type NOW-NOW

```
module WRCONFLCT2(bool &o) {
   next(o) = 1;
   next(o) = 2;
   pause;
}
```

Figure 2.5: Write conflict, type Typ NEXT-NEXT

types of write conflicts are caused by the same reason: two or more variables assign different values to one variable.

Shizophrenia Problems

These problems only occur in synchronous languages. In synchronous languages it is possible to execute a statement more than once in a macro step. Figure 2.7 shows an example for a shizophrenia problem [26]. To figure out the crucial point, assume that the control is at *l1*, and `i` holds. First, `a = true;` is executed. Due to the `abort` condition the second conditional statement is skipped. The next iteration of the loop start immediately and the first conditional statement is skipped; therefore, `a = true;` is executed again. However, `a` is assigned the same

```
module WRCONFLCT3(bool &o) {
   next(o) = 2;
   pause;
   o = 1;
}
```

Figure 2.6: Write conflict, type NEXT-NOW

```
module Schizo1(bool i, &a) {
   do {
      weak abort {
         if(!i) l1 :  pause;
         a = true;
         if(i) l2 :  pause;
      } when(i)
   } while(true);
}
```

Figure 2.7: Example for the shizophrenia problem.

value; therefore, this is no problem.
However, this can result in a problem, if `a` is a local variable. Figure 2.8 [26] shows the simplest example for reincarnation. Assume that the control is at *l1*. `x` is emitted; therefore, the second

```
module Reincar1(bool &xOn, &xOff) {
   do {
      local x;
      if (x) { xOn=true; } else { xOff=true; }
      l1 :  pause;
      x = true;
      if (x) { xOn=true; } else { xOff=true; }
   } while(true);
}
```

Figure 2.8: Example for the shizophrenia problem.

conditional statement emits `xOn`. The next iteration of the loop is started immediately and a new scope of local is entered, initializing `x` to zero. For that reason, the first conditional emits `xOff` and the control stops at *l1*. Hence, `xOn` and `xOff` are present in one macrostep.

Synthesis

To synthesize synchronous programs, they are usually compiled to intermediate formats.

Equational code [22]: The generation of equational code is convenient for the creation of hardware circuits. Additionally, it does not need to be sequentialized, which is an advantage for software synthesis. However, the whole code is always executed, even for idle sections. For that reason, the equational code runs very slow.

Job code: [24], [23] Job code is also an intermediate format for synchronous languages. It is intended to be used on multicore and multiprocessor architectures.

Automaton code: The representation of a synchronous program as an automata has the advantage that very efficient code can be created. However, the size of the automata increase exponentially with the size of the program's state space.

Averest intermediate format (AIF): The Quartz compiler creates a format that describes the behavior of the module using guarded actions, which are explained in the next section. This is the input format for the synthesis tool, presented in this thesis.

2.2 Guarded Actions

The guarded actions represent the input of the synthesis process; therefore, this section gives a basic overview. As already mentioned, one of the available output formats of the Quartz language is the Averest intermediate format (AIF). The AIF contains co-called guarded actions [22], [3], [8].

Definition 2 *A guarded action $\mathcal{A}$ for the variable x is defined by the tuple (φ, E) consisting of a guard φ and an expression E.*

Care has to be taken of the distinction between immediate guarded actions $\mathcal{A}_I$ and delayed guarded actions $\mathcal{A}_D$. Section 3 will explain the semantics of guarded actions in detail. In the following, it suffices to consider only immediate actions. Actually, the actions are the expressions on the right side of the assignments that occur in the Quartz description of the module. The expression is extended by a guard that defines the condition when the expression has to be assigned to the destination variable x. This guard depends on the control flow variables, and actually it codes the label where the control flow must be to execute the corresponding assignment. To given an example, consider Figure 2.9, which shows a simple Quartz module on the left side and its guarded actions on the right side. As can be seen, the assignment of

```
module Simple(bool i, int &o1, &o2) {
  l1:  pause;
  o1 = 100;
  l2:  pause;
  if(i) { o2 = 10 + o1; }
}
```

Guarded action for variable o1:
$\mathcal{A}_1 = (\mathtt{l1}, 100)$
Guarded action for variable o2:
$\mathcal{A}_2 = (\mathtt{l2} \wedge \mathtt{i}, 10 + \mathtt{l1})$

Figure 2.9: A simple Quartz module and the generated guarded actions.

100 to `o1` must be executed if the control is in `l1`. For that reason, the action's guard is `l1` and the expression is 100. The action for `o1` is defined similar. The expression on the left side, i. e. $10 + $ `l1` must be assigned if the control flow is in `l1`, and due to the conditional statement, `i` must hold.
In general, the translation of synchronous programs to guarded actions is not as easy as shown in this example. Schneider gives in [26] and [22] logical transition rules to translate statements of a synchronous program to a transition system. This transition system is used to build the guarded actions of the module. However, shizophrenia problems, i. e. the multiple execution of a Quartz statement in one macro step (see Section 2.1), and statements like the `abort` statement require the introduction of the so-called surface and depth of a module. A detailed description of this problem is omitted, because it requires detailed knowledge of the compilation using the logical transition rules. The cruicial point of this problem is that a statement may be instantaneous and not instantaneous, i. e. it may consume time or not, depending on the control flow. Therefore, one has to distinguish between the surface, which is executed when entering a statement, and the depth, which is executed if the control flow stays in the statement.
Beside guarded actions, a variable may also own a reaction to absence. The assignment of the reaction to absence is to describe the reset of event variables. The reaction to absence may also be used in the modular compilation of Quartz programs, which is not done in this thesis. However, for the sake of future changes, the reaction to absence is considered and handled in this thesis. Actually, the reaction to absence is also an action but with an implicitly defined guard. The reaction to absence has to be evaluated and to be assigned to the destination variable, if in the current macro step no immediate action was fired, and if in the previous macro step no delayed action was fired.

2.3 Cyclic Dependencies

To motivate the usefulness of cyclic dependencies, consider the definition of function z from [18]: $z =$`if` c `then` $F(G(x))$ `else` $G(F(x))$.

This function can be implemented in two different ways. The first possibility is to translate the expression in the definition without any changes. The resulting hardware implementation is shown in Figure 2.10 on the left side. Although one instance of each function is active, two instances for each function are necessary. There is a second possibility to circumvent the waste of resources and to build a more compact circuit. The hardware implementation is shown in Figure 2.10 on the right side. To be able to implement z with only one instance of each function, we implement the auxiliary functions y_F and y_G and modify z as follows:

$$\begin{aligned} y_F &= \texttt{if } c \texttt{ then } F(y_G) \texttt{ else } F(x) \\ y_G &= \texttt{if } c \texttt{ then } G(x) \texttt{ else } G(y_F) \\ z &= \texttt{if } c \texttt{ then } y_F \texttt{ else } y_G \end{aligned}$$

It is important to recognize that there is no logical feedback, although the circuit has a structural feedback. A description of the circuit in Quartz might look as shown in Figure 2.11.

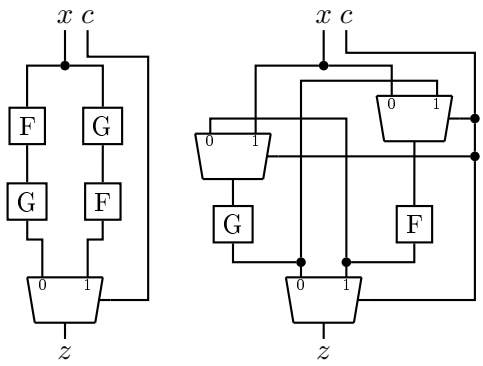


Figure 2.10: The function z=`if` c `then` $F(G(x))$ `else` $G(F(x))$. The left circuit is acyclic but uses two instances of each function. The right circuit needs only one instance of each function but has cyclic dependencies.

The compilation of the Quartz program will produce an AIF file that contains the actions shown in Figure 2.12.

```
module CyclicCombCircuit(bool c, int x, int &z) {
   local yF, yG
   while(true) {
      z = (c ?  yF : yG);
      yF = (c ?  (F(yG)) : (F(x)));
      yG = (c ?  (G(x)) :  (G(yF)));
      pause;
   }
}
```

Figure 2.11: Implementation of the cyclic combinational circuit in Quartz.

Malik presents a solution for constructing the outputs of cyclic circuits [18]. His solution applies three-valued logic to the circuit and requires an iterative computation of the outputs. However, three-valued logic introduces additional variables, i. e. in the worst case it doubles the variable space, and the additional computation effort is also a disadvantage.
In this diploma thesis, two solutions are given to solve this problem. The first solution uses partial evaluation which is introduced in [19] by Jones, Gomard, Sestoft. The partial evaluation will be explained in more detail in Section 4.2. The following gives a coarse overview of its application. The usage of partial evaluation is able to break up the cyclic circuit by setting the value of a variable to a constant. In general, this has to be done for all values that can be assigned to the variable. In this example, the synthesis tool will generate two blocks of code, one block for $c = false$ and one block for $c = true$. Applying the partial evaluation to the actions shown in Figure 2.12 yields the simplified actions shown in 2.13. Each subblock of the

$\mathcal{A}_1$:if c then $z = yF$
$\mathcal{A}_2$:if $\neg c$ then $z = yG$
$\mathcal{A}_3$:if c then $yF = F(yG)$
$\mathcal{A}_4$:if $\neg c$ then $yF = F(x)$
$\mathcal{A}_5$:if c then $yG = G(x)$
$\mathcal{A}_6$:if $\neg c$ then $yG = G(yF)$

Figure 2.12: The actions, which are generated by the Quartz compiler.

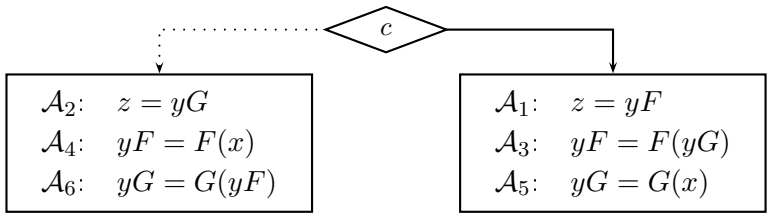


Figure 2.13: The simplified actions using partial evaluation.

partial evaluation is now free of cyclic dependencies; therefore, it can be easily sequentialized. The second solution is presented in detail in 4.4. In fact, the detailed explanation requires more knowledge of actions and the firing rule. Basically, the sequentializing algorithm has to find an order, so that no action reads a variable that is written by a succeeding action. Due to the guard of the actions, it is necessary to consider if a variable can be written under a condition. Paying attention to these conditions, it is possible to find a static schedule.

2.4 VLIW

Although, most modern processors have a superscalar architectures with dynamic scheduling, these processors have the disadvantage that they determine dependencies between instructions dynamically at runtime. This requires complex hardware circuits, and the examination of data dependencies in loops is actually always the same work.
To avoid these examinations of data dependencies, the Very Large Instruction Word architecture [28] leaves this work to the compiler. The compiler needs detailed knownledge of the target architecture to exploit the maximum performance. In general, a compiler has more time and more memory. This allows the compiler to apply better optimization algorithms, and to examine the code over large regions, i. e. it can apply global optimizations. However, some dependencies can only be checked at runtime, e. g. if two pointers are equal.
Beside the computations, the removal of the complex logic for dynamic scheduling can yield a significant reduction of power consumption. This is more and more a challenging topic in the development of processors. The reduced power consumption allows to implement more computational units, e. g. ALU or multipliers, onto the processor. This gives a potential increase in the number of instructions that can be issued per cycle.

3 AIF Input

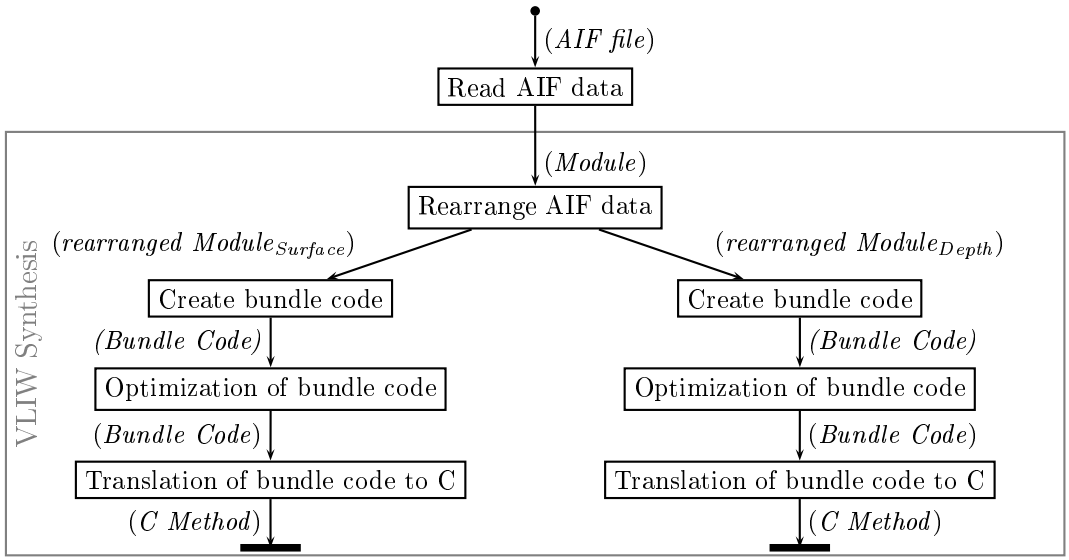


Figure 3.1: The different steps of the VLIW synthesis process

The synthesis process of an AIF file is shown in Figure 3.1. As can be seen, the VLIW synthesis process is split into two paths with four tasks on each path.
Reading the AIF file precedes the VLIW synthesis process and is given in a framework. The functionality of the AIF reader is simply mechanical and, therefore, it is not explained in detail. It reads the data from the AIF file and returns a data structure, called *module*, which contains all information about the module. It is important to know the data structure because it is used as input for the VLIW synthesis. The output of the AIF reader, i. e. the input of the VLIW synthesis, is explained in Section 3.1.
Having determined the input for the VLIW synthesis, the core of the synthesis process can start. The *module* structure is not the optimal representation of the module's information, therefore, it is reorganized by the first step of the synthesis. Due to the nature of a module, it has to be split into two parts, which are separately synthesized by the same algorithms. The reason for splitting the module is explained in Section 3.1. Each transformed input structure is then given to the bundle code creator. This task sequentializes the actions of the module according to their data dependencies. The output structure is basically a sequential list of instruction bundles. Each instruction bundle contains a set of instructions which are data independent and, therefore, they can be executed in parallel. The input format and the operation of the bundle code creator permit further optimizations of the bundle code. This optimization is done in the third task of the VLIW synthesis process - in the optimization

task, which is explained in Section 5. It could also be done in the bundle code creation task, but the separation of this task keeps the algorithm and the final implementation clear and understandable, respectively readable. The main goal of the optimization is to remove needless instructions to minimize the computation effort of the program at runtime. As a result of instruction removal, the remaining instructions might be rearranged to maximize the parallelism. The result of the optimization is again bundle code and is given to the last task in the VLIW synthesis process. The objective of the translation to C is to translate the bundle code is translated into C instructions and operations. The generated code is finally written to a file finishing the synthesis process for the specific part of the module. The synthesis of the module's actions into two methods and a framework completes the synthesis process.
Having finalized the overview of the synthesis process, a detailed description of each task can be started. The next section deals with the data structure of the AIF reader.

3.1 The AIF Input

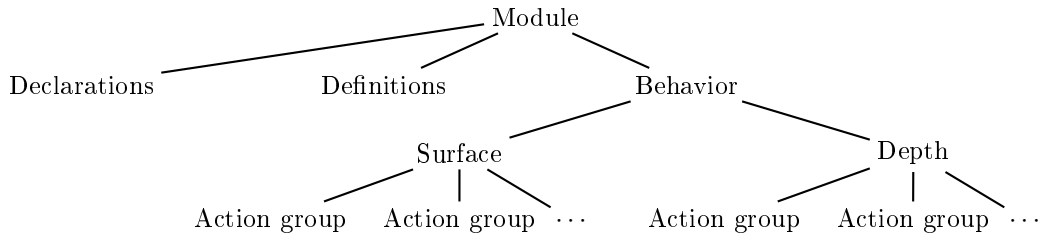


Figure 3.2: The structure of the input from the AIF reader

The module data structure collects the information about a module. This information is stored either in lists or other substructures (see Figure 3.2). This section explains the module data structure.
Basically, the AIF input that is necessary for the VLIW synthesis, consists of four lists: to begin with, the first list contains the declarations of the variables of the program. This includes inputs, outputs, but also local variables. First, the AIF module also contains so-called labels, which are used to implement the control flow of the program. A declaration consists of different components that describe how the variable has to be handled. Before describing a declaration, its components have to be defined.

Definition 3 *The interface type of a variable can either be input, output, local or label.*

Definition 4 *The data type of a variable defines the variable's representation in memory and the semantics of the operations that can be performed on it. The following type are available: Booleans, bitvectors, natural and integer numbers. The data type for labels is always Boolean and, therefore, it is not explicit given by an element in the structure.*

In fact, the AIF format provides more than these four types. Additionally, input variables are distinguished in controlled and uncontrolled inputs. The distinction of the inputs is not of

interest. The synthesis assumes that both types of input are always known at the beginning of a cycle and do not change during the calculation of the current cycle. Therefore, both types of input are treated as one type of input without distinguishing between controlled and uncontrolled input. Finally, all provided types can be grouped into the four classes given in Definition 3. If the variable's interface type is output or local, it has also a storage type.

Definition 5
- *The storage type of a variable is either memorized or event.*
- *The storage type for label variables is not given because it is always memorized.*
- *A memorized variable stores an assigned value until it is changed by another assignment.*
- *An event variable stores an assigned value for only one cycle. At the begin of the succeeding cycle, this value is invalidated, and the variable is assigned its default value unless another assignment is done to the variable.*

As can be seen, the storage type defines if a writeable variable stores a value or if it emits its value like an impulse, i. e. only for one cycle. Having defined the basic types of a declaration, the declaration itself can be defined.

Definition 6 *A declaration for a variable x consists of the variable's data type, its storage type and its interface type.*

The second list of the module data structure contains the so-called AIF definitions.

Definition 7 *An AIF definition (x, E) assigns the temporary variable x in the current cycle the value of the expression E.*

This temporary variable can be used in any other expressions. The background of AIF definitions in a module is based on the collection of shared expressions. During the compilation of a Quartz program, the compiler collects these shared expressions and adds them as a separate list to the module description. These AIF definitions can be handled in two different ways: the simplest solution is to replace all occurences of the temporary variables by their corresponding expressions. However, this method of handling the AIF definitions would completely remove the benefit of AIF definitions; therefore, it this is not done. A smarter but still not optimal solution is explained in Section 3.2. To be able to understand the solution, the concept of actions has to be defined. Actions are used in the third parameter of the module data structure, the behavior, and are explained in the following part. The declaration of the variables and the AIF definitions of shared expressions are the first important parts of the module. To complete it, the module's behavior has to be defined. The behavior is based on operations on the module's variables - the guarded actions, which were already defined in Definition 2. Of course, input variables can not be written by the module, i. e. only output, local and label variables have guarded actions. Moreover, Definition 2 does not define the behavior of the guarded action. Definition 10, which specifies the condition when an action has to be fired, is preceded by two definitions. First, a helper function has to be defined to determine which variables are necessary to evaluate an expression.

Definition 8 $\Delta(\varphi) = \{x | x \text{ appears in } \varphi\}$

The function $\Delta(\varphi)$ returns a list of all variables that are read by the formula φ. This includes output, local and label variables but also temporary variables of AIF definitions. The second definition is an anticipation of the notion of known variables. It is more an intuitive description than a precise definition. It will be defined more precisely by Definition 37, but for now, the following one sufficies.

Definition 9 *A variable is known iff its value in the current cycle can not be changed.*

Having assigned the function $\Delta(\varphi)$ and the notion of known variables, the condition for firing a guarded action can be specified.

Definition 10 *An guarded action $\mathcal{A} = (\varphi, E)$ for the variable x has to be fired iff the variables $\Delta(\varphi) \bigcap \Delta(E)$ are known, and φ can be evaluated to true.*

Hence, the following definition will help to get better bundle code by an early evaluation of the guard. If the guard can be evaluated to false, it is not necessary to wait for the availabilty of the input variables of E.

Definition 11 *A guarded action $\mathcal{A} = (\varphi, E)$ for the variable x has to be discarded iff the variables $\Delta(\varphi)$ are known and φ can be evaluated to $false$.*

In the next step, it has to be determined, what to do if an action has to be fired. The exact semantics of firing an action $\mathcal{A}$ depends on wether it is an immediate action or a delayed action. An immediate action $\mathcal{A}_I = (\varphi, E)$ for the variable x has the following semantics:

Definition 12 *Let $\mathcal{A}_I = (\varphi, E)$ be an immediate action for the variable x. Iff the firing condition holds, then x has to be assigned immediately the result of the evaluation of E.*

A delayed action $\mathcal{A}_D = (\varphi, E)$ for the variable x has the following semantics:

Definition 13 *Let $\mathcal{A}_D = (\varphi, E)$ be a delayed action for the variable x. Iff the firing condition holds then x has to be assigned in the next cycle the result of the immediate evaluation of E.*

A variable x can, but may not own an absence reaction. If x has an absence reaction and has not been assigned a value in the current cycle, i. e. in the current cycle no immediate action has fired and in the previous cycle no delayed action has fired, it has to be assigned the result of the evaluation of the absence reaction's expression.

Definition 14 *An absence reaction E_{abRe} for the variable x is an optional expression. If x has an absence reaction, E_{abRe} has the same type as x, else E_{abRe} has the value $NONE$.*

Since an absence reaction has no guard, its firing condition differs from the firing condition for actions by the following rule.

Definition 15 *Let E_{abRe} be the absence reaction for x and $E_{abRe} \neq NONE$. Iff in the current cycle no immediate action of x has been fired and in the previous cycle no delayed action of x has been fired, the variable x has to be assigned immediately the result of the evaluation of E_{abRe}.*

Having defined the basic elements of the AIF input data and their semantics, they can be grouped to an action group.

Definition 16 *An action group assigns a destination variable x a list of immediate actions $\mathcal{A}_{I,1}, \mathcal{A}_{I,2}, \ldots, \mathcal{A}_{I,n}$, a list of delayed actions $\mathcal{A}_{D,1}, \mathcal{A}_{D,2}, \ldots, \mathcal{A}_{D,l}$ and an optional absence reaction E_{abRe}. An action group is represented by the following tuple:*
$(x, \{\mathcal{A}_{I,1}, \mathcal{A}_{I,2}, \ldots, \mathcal{A}_{I,n}\}, \{\mathcal{A}_{D,1}, \mathcal{A}_{D,2}, \ldots, \mathcal{A}_{D,l}\}, E_{abRe})$

At this point, it is important to discuss write conflicts. Output and local variables can have more than one immediate action or delayed action. Reconsider that actions in synchronous languages are executed in parallel and therefore, they have no order. Furthermore, the guards of the actions of one variable do not have to be disjunctive, which introduces the write conflicts. A write conflict is present iff two or more actions of the same variable and the same type are fired in a cycle. Write conflicts cause an undefined behavior of the module. This also applies to the sequentialized code of the module. In Averest, the absence of write conflicts can be proved by model checking. This is not part of this diploma thesis; therefore, one can assume that the module is free of write conflicts - otherwise the module and also the synthesized code would not make much sense. To define the absence of write conflicts, an anticipation of the expression reachable states has to be made.
In general, every writeable variable can be assigned any value of its type's range, e.g. a Boolean variable can be *true* or $false$. Hence, every configuration of the variables is possible, but not every configuration of the variables makes sense. Consider the expression $o = i$. One can easily see that only the configuration $o \wedge i$ or $\neg o \wedge \neg i$ can be reached. In this case $o \wedge i \vee \neg o \wedge \neg i$ would be the symbolic representation of reachable states. This topic is explained later in more detail (see Section 4.3).

Definition 17 *Let r be a symbolic representation of all reachable states. A module has no write conflicts iff for all writable x with action group*
$(x, \{(\varphi_{I,i}, E_{I,i}) | i \in 1 \ldots n\}, \{(\varphi_{D,i}, E_{D,i}) | i \in 1 \ldots l\}, E_{abRe})$*, one has:*
$(\forall i, j \in \{1, \ldots, n\}, i \neq j.(\varphi_{I,i} \wedge \varphi_{I,j} \wedge r) = false) \wedge$
$(\forall i, j \in \{1, \ldots, l\}, i \neq j.(\varphi_{D,i} \wedge \varphi_{D,j} \wedge r) = false) \wedge$
$(\forall i, j \in \{1, \ldots, l\}, i \neq j.(((\varphi_{I,i} in\ cycle\ l{+}1) \wedge (\varphi_{D,j} in\ cycle\ l) \wedge r) = false)$

In Section 2.1, three different types of write conflicts were presented; therefore, the absence of write conflicts requires to comply with three conditions: the first condition ensures the absence of NOW-NOW write conflicts, the second condition ensures the absence of NEXT-NEXT write conflicts and the last condition exclude a write conflict by an immediate and a delayed assignment, i. e. the absence of NEXT-NOW write conflicts.

In general, a module contains more than one output variable, so the AIF reader will return a list of action groups. In particular, it will return two lists. One list groups the so-called surface of the module and the second groups the so-called depth of the module.

Definition 18 *The surface of a module contains all actions that can be executed iff the module is entered.*

Definition 19 *The depth of a module contains all actions that can be executed iff the control flow stays in the module, i. e. the module did not terminate since it has been entered.*

The synthesis only considers one module. This condition simplifies the definitions of the surface and the depth as follows:

Definition 20 *The surface of a single module contains the actions that can be executed at starting time.*

Definition 21 *The depth of a single module contains the actions that can be executed after starting time.*

The last two definitions express that the synthesized program needs two different methods to run a module. The surface is used to synthesize the initialization method for the module, while the depth contains the information to synthesize the main method, which is called in each cycle except for the first one. Hence, the synthesis process runs twice with a different set of actions in each run, i. e. the first run synthesizes the surface actions to the initialization method while the second run synthesizes the depth actions to the main method. Having defined the surface and the depth of a module, an important supplement has to be done. Both, the surface and the depth are split into a control flow and a data flow.

Definition 22 *The control flow contains the actions that manipulate the label variables of the module.*

A remark has to be done about this definition: it has to be taken care about that a macro step is not a single step in the synthesized program. It is possible to execute several macro steps in a single cycle.

Definition 23 *The data flow contains the actions that manipulate the local and output variables of the module.*

For synthesis, there is no necessity to distinguish between firing an action of the control flow and firing an action of the data flow. The only difference is that the action group for a control flow variable will always contain delayed actions only, but no immediate action and no absence reaction. Due to this fact, the control flow and the data flow are concatenated so that the surface and the depth are each represented by one list of actions.

3.2 Rearranging the Module

AIF definitions have been explained in Section 3.1, but their handling is still undefined. Like guarded actions, the AIF definitions can not be executed in an arbitrary order. They have to wait for the availability of the input variables, too. This leads to the firing condition of the AIF definitions which is similar to the firing condition of the guarded actions.

Definition 24 *An AIF definition (x, E) for the variable x can be fired iff the variables $\Delta(E)$ are known .*

As can be seen, this definition does not force the firing of an AIF definition. The firing condition of an AIF definition also depends on its usage. Section 4.4 will give a detailed explanation for determining the exact firing condition. The exact firing condition is necessary to break up cyclic dependencies. However, it may delay the firing of an AIF definition. For a first explanation of the synthesis process, the exact firing condition is omitted. Due to the intention to fire actions as early as possible, the firing condition in Definition 24 is intensified to *must be fired*. Having defined the firing condition, the next step is to define the behavior of the AIF definitions.

Definition 25 *Let (x, E) be an AIF definition. Iff the firing condition holds, then x has to be assigned immediately the result of the evaluation of E.*

Taking a close look to Definitions 10, 24, 12 and 25, one can see that the behavior of AIF definitions is similar to that of immediate guarded actions. AIF definitions are a special case of local variables with exactly one immediate action and no delayed actions and no absence reaction. A question that arises now is the determination of a guard for the immediate action. Intuitively, the guard has to be set to *true*, because an AIF definition does not depend on the control flow. Additionally, an always satisfying guard does not delay the firing of an AIF definition; therefore, it meets the requirement that an AIF definition must be fired as early as possible. The solution which suggests itself is to handle AIF definitions like guarded actions. Hence, this is a convenient solution for the implementation of the bundle code creation task because the changes concern only the rearranging of the module. However, transforming AIF definitions to guarded actions introduces the necessity of optimizations. The explanation of this point follows the next definition. The first step for handling AIF definitions is to create an action group for each AIF definition.

Definition 26 *The action group for an AIF definition (x, E) is represented by the tuple $(x, \{(true, E)\}, \{\}, NONE)$.*

The new list of action groups must not be appended to the conventional list of action groups, neither the surface nor the depth. Although, the AIF definitions can be handled like conventional guarded actions, one important difference still remains. The destination variable of an AIF definition does not have to carry its value of a cycle to the succeeding cycle, i. e. a value is only valid in the cycle when it is calculated. Handling AIF definitions like 'real' local variables is not wrong but it is not optimal, either. Creating the bundle code can be done by the suggested solution, i. e. AIF definitions are handled like local variables with one immediate guarded action. But it is strongly recommended to optimize the created bundle code afterwards (see Figure 3.1). To be able to optimize the bundle code after its creation, all assignments to an AIF definition variable have to be marked.
Having defined the handling of AIF definitions, the next task is to find a convenient representation of the data that is necessary to build the bundle code. In the next step of the VLIW synthesis process, this bundle code will be generated by firing the actions one after the other. The fired actions will be translated to assignments which are the basis of the bundle code. During this translation process, some information has to be checked again and again. To avoid

unnecessary computation effort by recalculating these values, the action group of a variable is collected with all needed values and constants to one data structure - the so-called **action guide** - which has the following members:

dataType this is the data type of the destination variable. If a variable gets known according to Definition 9 it is appended to the list of known variables. Section 4.2 will explain the functionality of partial evaluation which can be extended according to the description given in Section 4.4. In fact, a variable that gets known can also be used for partial evaluation. However, partial evaluation can only be applied to variables of a specific type. *dataType* is to determine if a variable that gets known has to be appended to the list of known variables or to be used for further partial evaluation.

storageType this is the storage type of the destination variable. It is used for the translation of the sequentialized actions to C code. Events have to be reset after one cycle which is not handled by the sequentialization algorithm. Resetting events is done by additional instructions that are generated in the translation task.

actionGroup this is the group of the variable's actions which can be split into the destination variable, a list of immediate actions which may be empty, a list of delayed actions which may be empty and an optional absence reaction.

canBeWrittenDelayed states if the variable can be written by a delayed action. This value is initialized once and not changed, i. e. it does not show if the variable can still be written by a remaining delayed action. This flag is intended to optimize the firing of absence reactions. Due to the current approach of the synthesis process this can not be included yet. The changes that have to be done to implement the optimization can be done in a later step of the implementation. Anyway, the optimization increases the performance of the program between hardly ever and slightly. Therefore, the mentioned optimization is explained in a later Section 4.4 to avoid confusions with details.

overallGuardImmediate the overall guard is used to build the disjunction of the guards of all fired immediate actions. If the overall guard of immediate actions can be evaluated to `true`, all remaining immediate actions and the absence reaction can be discarded due to the assumption that there are no write conflicts.

hasBeenWritten is a state for observing if the variable has been written. The state can have the following values:
`has_not_been_written`,
`might_have_been_written` or
`has_been_written`
It has the same purpose as *overallGuardImmediate*, but expresses the result in an intuitive verbalised way.

isDefinition if the destination variable of this structure is an AIF definition variable then *isDefinition* is set to `true`. After sequentializing the concurrent actions the AIF defini-

tion assignments in the generated bundle code can be optimized. This optimization is explained in Section 5.

The representation of the module's behavior is now clear. This is done by the action guides explained in the last part of this section. Furthermore, in general a set of known variables V_{known} is necessary to fire at least one action. Considering Definition 9, known variables are variables that will not change their value in the current cycle. At the beginning of a cycle, these variables will be those that cannot be written immediate or by an absence reaction, i. e. known variables are all input variables and all output variables that can only be written by delayed actions.
Furthermore, the bundle creation task uses partial evaluation to reduce the code that has to be executed per cycle. Section 4 gives a detailed description of the partial evaluation. Here, only the fundamentals will be introduced. The goal is to reduce the number of assignments that have to be executed in a cycle. During the sequentialization process, guarded actions have to be fired. Recall that the guards consist of variables that are known according to Definition 9, but their values are not known. Hence, guarded actions are synthesized to conditional assignments. In the worst case, i. e. if no action has a guard that is true, for every guarded action a conditional assignment is created. To reduce the number of assignments, the evaluation of the guards has to be improved. Instead of regarding the general case for a variable, for each possible value than can be assigned to the variable, a partial evaluation is created for the generated bundle code. For each case, all appearances of the variable in an expression or a guard are replaced by the corresponding value. Using this replacement strategy and applying lazy evaluation to the simplified expressions including the guards, it is possible to reduce the size of the expressions. The optimal case is that a guard can be completely simplfified to false or true, i. e. the action can be removed or fired without a condition.
After the short introduction to the partial evaluation, it has to be declared which variables have to be used for it. Although, it is possible to use the partial evaluation with each variable that is known according to Definition 9, it does not mean that it is reasonable to do that. Recall that the goal is to simplify the guards of the actions. The guards usually consist mostly of label variables, therefore, the obvious choice is to use label variables for the partial evaluation. As described, these variables have to be known, which leads to the method that creates the list of variables that are used for partial evaluation. All label variables in the list of known variables V_{known} are moved to a new list V_{CD}, which is one of the parameter for the bundle code creation task. Labels are usually written delayed-only which implies that they could be copied directly to the list for partial evaluation. However working with this indirection has the advantage that it is safe for future changes to this condition. As already mentioned, it is possible to apply the partial evaluation to any known variable. Hence, the decision which variable is used for this technique is left to a simple filter, which is applied to the list of known variables. To give an example, the filter can be chosen depending on the size of the list of known variables. For small modules, the filter will select all Boolean variables. In contrast to that, only specific variables will be selected for large modules.

4 Sequentialization

4.1 Bundle Code Format

The bundle code creation can also be described as the sequentialization process. The basic approach of the sequentialization algorithm presented in this diploma thesis is based on the application of the firing rules presented in Section 3.1. The main goal is to find a static schedule for executing the module's guarded actions. Before bundle code can be created, the format and its meaning has to be described. The basis for the sequentialization algorithm is a list of action guides, which is created by the rearranging task. This task was already described in Section 3.2. The action guides contain the guarded actions that build the data flow graph and the control flow graph. Again, they represent the data dependency graph of the module, which controls the firing sequence of the actions.
The synthesis is however different to the simulation of a data flow computer. First, there are methods produce different results. The synthesis creates a static schedule of the concurrent actions while the simulation runs the program directly. The second point is that the synthesis contains variables, i. e. values are stored permanently and not temporary on a virtual edge. The third and most important difference is an advantage that may result of the partial evaluation. Not all variables in an expression have to be known to compute the final result. A simple example is a C-like `if-then-else` expression $\psi ? E_{true} : E_{false}$. If ψ is evaluated to $true$ or $false$, then it is known which expression has to be evaluated and which expression can be discarded. In principle, data-flow computers know data-driven and demand-driven executions; therefore, it has to regard all actions in a cycle. In contrast to this technique, the synthesis may create code that is optimized to specific states and, therefore, not every action has to be regarded. The last point in that the two techniques differ concerns the definitions that are used in AIF descriptions to collect shared expressions. Although they have already been transformed to actions, i. e. they have been integrated into the data flow graph, the static schedule can be optimized. It was already mentioned that the bundle creation task is followed by a optimization step, i. e. this optimization needs not to be handled in this section.
Before giving a detailed description of the algorithm, a clear description of the bundle code's data structure is needed. The bundle code is a list of bundles where the end of the list can be the end of the program or a fork, created by the partial evaluation. The bundle code can be graphically represented as a tree, treating lists of bundles as leafs or as nodes with exact one successor and treating forks as nodes with at least one successor. The format of the bundle

code can be easily seen by taking a look to the BNF definition of the bundle code.

<bundle code>	::=	<bundle list>? <partial evaluation>?
<bundle list>	::=	(<bundle>)+
<bundle>	::=	(<assignment>)+
<assignment>	::=	<immediate assignment> |
		<delayed assignment> |
		<absence assignment>
<immediate assignment>	::=	(<variable>, ϕ, <expression>)
<delayed assignment>	::=	(<variable>, ϕ, <expression>)
<absence assignment>	::=	(<variable>, c, <expression>)
<partial evaluation>	::=	(φ, <bundle code>$_{\text{then}}$, <bundle code>$_{\text{else}}$)

The three different assignment types, the immediate, delayed and absence assignment, correspond to a fired action of the same type. The BNF for <variable> and <expression> is not given in this context because they are only used in the BNF definitions of the assignments. These assignments and their behavior are defined as follows:

Definition 27 *An immediate assignment $\mathcal{I}_I = im(x, \varphi, E)$ contains a destination variable x, a condition φ and an expression E.*

The next definition describes the execution of an immediate assignment.

Definition 28 *Let $\mathcal{I}_I = im(x, \varphi, E)$ be an immediate assignment. Iff the condition φ is evaluated to true, the destination variable x has to be assigned immediately the result of the evaluation of E.*

The next step is to define the delayed assignment and its behavior when it has to be executed.

Definition 29 *A delayed assignment $\mathcal{I}_D = de(x, \varphi, E)$ contains a destination variable x, a condition φ and an expression E.*

Definition 30 *Let $\mathcal{I}_D = de(x, \varphi, E)$ be a delayed assignment. Iff the condition φ is evaluated to true, at the beginning of the next cycle the destination variable x has to be assigned the result of the immediate evaluation of E.*

The semantics of the delayed assignment is similar but not equivalent to the semantics of the immediate assignment. Special care has to be taken of the point of time when the expression E has to be evaluated and when it has to be assigned. The evaluation of E has to be done immediately, i. e. in the same cycle, but the result must be assigned to x at the beginning of the next cycle.

Definition 31 *An absence reaction assignment $\mathcal{I}_{abRe} = abRe(x, c, E)$ contains a destination variable x, a state c and an expression E.*

Definition 32 *Let $\mathcal{I}_{abRe} = abRe(c, x, E)$ be an absence reaction assignment. x has to be assigned immediately the result of the evaluation of E, iff c is false or in the current cycle, neither an immediate assignment nor a delayed assignment has been done to x.*

Obviously, the absence reaction assignment must not be executed before any immediate assignment in the current cycle or a delayed assignment from the preceding cycle. As said in Definition 30, the assignment of a value to a variable that was triggered by a delayed assignment has to be done at the beginning of a cycle. Therefore, if a variable owns an absence reaction, the absence reaction assignment is always the last potential assignment to x during a cycle, therefore, it can be fired at the earliest after the last immediate assignment.
Although, a BNF definition of a bundle and a bundle list is given, they have to be defined more precisely. As can be seen in their BNF definition, a bundle list is basically a list of bundles. Then again, the bundles are a list of assignments. This implies that the bundle list itself is a list of assignments. This raises the question, why the bundle list is split into at least one sublist of assignments. This question is answered basically by the following explanation: a bundle contains a list of instructions that can be executed in parallel, while the instructions of different bundles have a data dependency, i. e. they have to be executed in a specific order. This implies that the bundle list has a defined execution order. This explanation is deepened by the following definitions.

Definition 33 *An assignment A_1 is data independent of assignment A_2 iff*

- *A_2 is a delayed assignment or*
- *(A_1 is an absence assignment $abE(c_1, x_1, E_1)$ and x_2 is the destination variable of the assignment A_2) $\rightarrow x_2 \notin \Delta(E_1)$ or*
- *(A_1 is an immediate assignment $im(x_1, \varphi_1, E_1)$ and x_2 is the destination variable of the assignment A_2) $\rightarrow x_2 \notin \Delta(\varphi) \wedge x_2 \notin \Delta(E_1)$ or*
- *(A_1 is an delayed assignment $de(x_1, \varphi_1, E_1)$ and x_2 is the destination variable of the assignment A_2) $\rightarrow x_2 \notin \Delta(\varphi) \wedge x_2 \notin \Delta(E_1)$*

With that knowledge one is able to define a group of independent assignments - the bundle.

Definition 34 *A **bundle** $\mathcal{B} = (assignment_1, assignment_2, \ldots, assignment_n)$ consists of arbitrary but finitely many independent assignments.*

This definition of a bundle enforces that the assignments that form the bundle can be executed concurrently or in an arbitrary order without changing the result. Usually, the number of assignments in a bundle is limited to the processor architecture, but the code synthesis abstracts from any architectural limits, i. e. the number of assignments in a bundle can be as large as the number of actions in the corresponding module. Additionally, one goal of the synthesis is to determine the maximum parallelism that a synchronous program can provide by sequentializing the actions.
The next definition concerns the execution of a bundle:

Definition 35 *Let $\mathcal{B} = (assignment_1, assignment_2, \ldots, assignment_n)$ be a bundle. $\mathcal{B}$ is executed by executing every assignment in the bundle. In particular, the execution of B starts with the execution of the first assignment in the set - reconsider that the order does not matter - and ends with finishing the execution of the last assignment.*

The detailed definition of bundles enables to define the bundle list:

Definition 36 *A **bundle list** is a list of bundles with a specified sequential execution order.* $\mathcal{B}_{list} = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_l\}$ *with* $\forall i = 1, \ldots, l. \forall j > i. \mathcal{B}_i$ *has to be executed before* $\mathcal{B}_j$

The previous definitions allow a precise definition of whether a variable is known or not.

Definition 37 *A variable* x *is known in a bundle* $\mathcal{B}_l \leftrightarrow \forall k \geq l. \neg(\exists \mathcal{I} \in \mathcal{B}_k, \varphi, c, E. \mathcal{I}_I = im(x, \varphi, E) \vee \mathcal{I}_{abRe} = abRe(x, c, E))$.

A variable x is known in a bundle iff this bundle and all following bundles contain neither immediate assignments nor an absence assignment to x, i. e. iff the variable x can not be changed by the bundle or any succeeding bundle. This applies to the intuitive Definition 9. The next section gives a detailed description of the goal and task of the partial evaluation which has already been mentioned in this section.

4.2 Partial Evaluation

A short introduction to the partial evaluation was already given in Section 3.2. Reconsider the format of the bundle code. Basically, it represents a static schedule of guarded actions. Obviously, the less actions are in a schedule the faster the program will run. This leads to the approach of the partial evaluation. In general, it is not possible to reduce the number of actions. Due to the guards which are finally the condition for firing an action, not every action is really fired. One could say that usually only a few of the guarded actions can fire in a cycle. This depends mainly on the configuration of the labels in a cycle. Hence, the goal is to reduce the number of actions for a specific schedule. Specific schedules, in this terms means a schedule with defined values for a set of variables. Hence, in general, a schedule has to be generated for each set of values that can be assigned to a set of variables, i. e. a program consists of several specific schedules. Each schedule can be synthesized to a fragment of the whole program. Each fragment can consist of one or more assignments and it is possible that for a specific constellation no action has to be executed, i. e. the corresponding fragment is empty. Finally, all fragments are connected by `if`-statements to a complete program. However, in general, the number of assignments that have to be executed in a step does not change, but the number of assignments that have to executed in a fragment can be reduced.
The usage of partial evaluation modifies the code in two points: first, actions with the same guard or with a partially similar guard are grouped, i. e. the guard, respectively the part of the guard, is checked only once. This leads to the second point that concerns the VLIW optimization. To explain the acceleration that can be achieved by the partial evaluation, consider the following example. Let l_1 and l_2 be some labels that are exclusive, i. e. at most one variable is set to *true*. Additionally, assume that action $\mathcal{A}_{i,j}$ is an assignment $\mathcal{I}_{i,j}$ with the predicate c_i. The semantics of the actions can be described as follows:

$$\forall j = 1 \ldots n. \forall i = 1 \ldots 3. \mathcal{A}_{i,j} = \texttt{if } c_i \texttt{ then } \mathcal{I}_{i,j}.$$

Hence, the data dependency can be seen from the bundles that are generated by the generic schedule (see Figure 4.1, left side). In particular, it is defined by

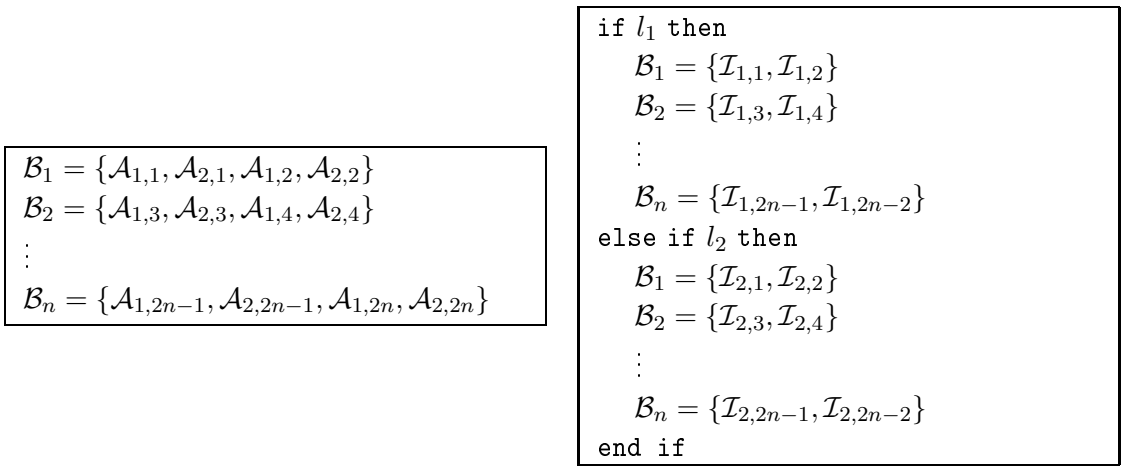


Figure 4.1: Left: a generic schedule for a module, right: the same module, but scheduled using partial evaluation

1. $\forall k = 1 \ldots n . \mathcal{A}_{1,k}, \mathcal{A}_{2,k}, \mathcal{A}_{3,k}$ *are data independent* and
2. $\forall l = 0 \ldots n-1 . \mathcal{A}_{1,3l+1}, \mathcal{A}_{1,3l+2}, \mathcal{A}_{1,3l+3}$ *are data independent.*

A generic schedule is shown on the left side of Figure 4.1. The right side shows the corresponding specific schedules for l_1 and l_2. As already mentioned, the label variables are exclusive. That implies that only one `if`-block is executed per cycle. As can be seen, the number of bundles that have to be executed in a cycle does not change, but their sizes have been halved. At this point, it is important to emphasize that in both methods, the number of assignments that have to be executed per bundle are equal. In the generic schedule, only two of four instructions are executed in a bundle due to the mutual excluding labels. Whereas, the specific schedule contains only two instructions that are always executed since the corresponding label has been checked before.
This optimization can be useful in two ways: as already mentioned, the reorganized specific schedule can contain less resources than the original schedule. Using a specific VLIW architecture with a limited number of instructions per bundle, the specific schedules lead to a smaller number of bundles. This is explained as follows: the bundles that are created in the bundle creation task are not limited in their size. They can have an arbitrary number of assignments. But in practice, the size of a bundle is limited by the processors architecture, e.g. the Intel Itanium takes three instructions per word. With reference to this example, for a VLIW processor that takes less than four instructions per word, each bundle in the generic schedule has to be split into two bundles. This results in a code that is twice as large as the original one and than the code of a specific schedule. Finally, this could end in a speed-up of the program.

A set of variables is needed for the partial evaluation. This set is created by the task that rearranges the module (see Figure 3.2). Now, the partial evaluation can be done in two slightly different ways. The first method makes an own partial evaluation for each variable while the

```
if ¬a then
   if ¬b then
      ...
   else
      ...
   endif
else
   if ¬b then
      ...
   else
      ...
   endif
endif
```

```
if (¬a ∧ ¬b) then
   ...
else if (¬a ∧ b) then
   ...
else if (a ∧ ¬b) then
   ...
else
   ...
endif
```

Figure 4.2: Two methods to apply the partial evaluation. (left: a separate partial evaluation for each variable; right: a partial evaluation for each state in the complete state space).

second method builds a partial evaluation for each state in the complete state space that is build by the variables for the partial evaluation. Figure 4.2 shows the abstracted results of a synthesized program with the labels a and b. The former method can be described as a binary search for the current state, whereas the latter one applies a linear search. Due to the lack of time only one method has been implemented. The choice was to use the first method, which is justified by following reason: two nested if-blocks may be "unrolled" quite easily by creating a copy of the lower-level if-block and adding the condition of the higher-level if-block to the first copy and adding the negated condition of the higher-level if-block to the second copy. This can be done recursively to an arbitrary number of nested if-blocks. Vice versa, creating nested if-blocks of a sequence of if-blocks with disjunctive conditions needs more effort due to the necessity of analysing the conditions. Let V be a set of variables that is intended for the partial evaluation. Assume that this set contains n variables, i. e. $|V| = n$. Consider the following two cases:

- First case: n is a small number, i. e. $n < 6$. Hence, the number of states is $2^n < 2^6 = 64$. At this point, it has to be emphasized that the final compilation is done by a compiler that is independent of the synthesis tool. The following observations can be made executing the synthesized program on a real processor. Considering the Intel Itanium as the target plattform for the program, all states could be stored in the Itanium's predicate registers. Hence, all instructions could be executed in parallel using the predicate registers. The final usage and application of predicate registers depends on the compiler and may depend one the number of instructions in the paths.

  - Assuming that each path consists of many instructions, we can reconsider the example presented in Figure 4.1. The processor usage will be high, but only a few of

the calculated results are applied. Again, it has to be mentioned that the goal is to use the benefit of VLIW processors, i. e. executing instructions in parallel, but not to waste it.

– However, if the paths consist of only a few instructions, the usage of predicated execution avoids conditional branches that might stall the pipeline due to mispredicted branches. In this case, the second presented method would be the better choice, but a check with the Intel IA-64 C++ compiler and the VEX compiler has shown that the same code is generated for both methods. This means that the optimizations that can be done by synthesizing code with the second method, can also be done with the first method.

Therefore, the first case does not influence the decision about the choice of the method.

- Second case: n is a large number, i. e. $n > 10$. The Intel Itanium processor owns 64 predicate register. By far, this is not enough to cover all cases. Moreover, for each case, an additional expression has to be generated to calculate one predicate register. Even, if a part of the partial evaluation can be done by predicated execution, some if-blocks still remain. As already mentioned, the first method that can be compared to a binary search, whereas the second method is a linear search for the current state. Therefore, the better choice is to use the first method. The second method might be of interest, if probabilities for each path are considered. Ordering the if-blocks according to their probability could improve the latter method

To summarize: the goal of partial evaluation is to avoid stupidly build code that wastes the benefits of VLIW processors. It is used to get a better schedule which makes a more efficient usage of the resources of a VLIW processor. A side effect is the potentially reduction of the number of assignments that have to be executed in a cycle and the potentially reduction of the number of bundles that are necessary in a specific schedule.

4.3 Reachable Sstates

The use of partial evaluation introduces a new problem: currently, it is necessary to build for each case a separately build schedule, i. e. bundle code is generated for each case. In particular, each Boolean variable that is used for partial evaluation doubles the state space; therefore, the number of cases that have to be considered are doubled, too. This emphasizes the disadvantage of the partial evaluation, which can lead to large code when synthesizing large modules. Of course, this can be avoided by limiting the maximal number of variables that are used for the partial evaluation. This solution is not satisfying; therefore, another method has to be found. In general, all possible states have to be considered. However, in some programs, it is not possible to reach every state. Hence, it is not necessary to create bundle code for these states. This section discusses this problem and gives a basic approach to get the reachable states.
Reconsider that synchronous languages have been defined to simplify the implementation of

parallel programs. Due to a specific timing, some processes, e.g. communication protocols, have to execute a group of instructions in a defined order, i. e. a sequential list of instructions. Therefore, it is usual to build sequences of macro steps in synchronous programs. Figure 4.3

```
module  WaitForI_2(event  i, event
&o) {
   loop {
      do {
      l1: pause;
      } until(i);
         emit o;
      l2: pause;
   }
}
```

Figure 4.3: The program `WaitForI` waits for the presence of i and emits o if it is set.

shows a Quartz implementations of a program that waits for a signal i and emits o if i is set. l_1 and l_2 are the control flow locations, i. e. they control which macrostep has to be executed. Basically, the first macro step waits for the presence of i, but waits at least one cycle. The second macro step simply emits o. An important point is that the program can not wait for i and emit o at the same time. This means that only one of the labels l_1 and l_2 can be active at the same time. Obviously, it is a good idea to generate code only for the reachable states which might reduce the size of the generated code.

Combining the Reachable States with the Partial Evaluation

Finally, the partial evaluation has to be restricted to the reachable states. Reconsider two points: first, the partial evaluation in this diploma thesis is applied to variables and not to the states (see Section 4.2). Second, the partial evaluation is done by replacing all occurrences of a variable by a constant value. To restrict the partial evaluations to the reachable states, this replacement has also to be done in the expression that contains all reachable states. If this reachable states expression is evaluated to false, the state is not reachable; therefore, no code has to be generated for this path. In particular, this means that the variable must not have the value that is assigned by the replacement. Then again, this builds a constraint for the variable, i. e. the variable must have the alternative value. Due to this reachable constraint, the if-block for this partial evaluation can be omitted. An alternative way to combine the reachable states with the partial evaluation is to enumerate all reachable states and to build a sequential list of if-blocks. This is similar to the second method presented in Section 4.2, i. e. it corresponds to a linear search and, therefore, it is rejected.
Anyway, the generated code of the previous described method may result in larger conditions. To explain that, reconsider the example in Figure 4.3. The control flow must stop either in l_1 or in l_2. Therefore, the reachable states are $l_1 \wedge \neg l_2 \vee \neg l_1 \wedge l_2$. The first method will make

```
if l1  then
   ...
else
   ...
end if
```

```
if l1 and (not l2) then
   ...
else if (not l1) and l2 then
   ...
end if
```

Figure 4.4: Left: partial evaluation of labels with reachable constraints, right: partial Evaluation of states with reachable constraints

a partial evaluation for l_1. In particular, a copy is created of all actions and the reachable states. In the first copy all occurrences of l_1 will be replaced by $true$, and in the second copy all occurrences of l_2 will be replaced by $false$. The actions will be now neglected. The remaining states will be $\neg l_2$ for the l_1-path, and l_2 for the $\neg l_1$-path, respectively. For each of these paths, the partial evaluation for l_2 is applied. In the l_1-path, this leads to the result that l_2 is not reachable, therefore, l_2 must be $false$ and then again, code will be generated only for the $\neg l_2$ path. Due to the constraint, no `if`-statement is necessary for this partial evaluation. The $\neg l_1$ path is analogous. The result is shown in Figure 4.4 on the left side. In contrast, the second method would create a list of states and make a partial evaluation for each state. The result of this approach is shown in Figure 4.4 on the right side.

Reachability Analysis

A detailed description of the calculation of the reachable states exceeds this thesis; therefore, only the basic procedure is given. However, this requires the knowledge of Kripke structures, symbolic representation and the application of a so-called fixpoint iteration. Furthermore, the module's guarded actions have to be translated to a single equational system. See [22], [6], [21], [29] for a detailed description of the calculation of the reachable states of guarded actions and the used methods.
The reachability analysis is done using Kripke structures [27]. A Kripke structure is a tuple $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, where $\mathcal{S}$ is a set of states, $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states, $\mathcal{R}$ is the transition relation and $\mathcal{L}$ is a so-called label function. $\mathcal{L}(s)$ returns the set of variables that hold in the state s. To calculate the reachable states, one needs the set of states, which is given by the module's variables. The initial states are given by the initial values of the module's control signals. Furthermore, the transition relation is required. In fact, the transition relation assignes a successor state to a state. In particular, a function calculates in dependence of the current configuration of the variables the configuration of the variables of the succeeding state. The transition relation is received as follows: the module's actions are translated into a single equational system according to [22], Section 5.2. The variables on the left side of the equational system are the successors while the variables on the right side contain the current values. Actually, the equational system represents the transition relation. Usually the sets are represented using symbolic descriptions [6]; therefore, the label function is not necessary. Having built the Kripke structure, one is able to calculate the reachable states. This is done as follows: one assigns $\upsilon_0 = \mathcal{I}$ and calculates υ_0 successor states by applying the transition

relation to it. υ_0 is added to its successor states yielding υ_1. In general, this procedure has to be repeated several times, i. e. the transition relation is applied to υ_n to gain the succesor states of υ_n. The union of these states and υ_n yields υ_{n+1}. Note that $\upsilon_n \subseteq \upsilon_{n+1} \subseteq \mathcal{S}$. A fixpoint iteration applied to this procedure yield the reachable states. The set of reachable states is represented using symbolic descriptions, i. e. as a Boolean formular. This is exactly the representation that is required in the following sections.

4.4 Sequentialization Algorithm

In the beginning of this section, the BNF of the bundle code format was presented. Until now, only the distinctions have been handled. Although, they can influence the number of actions that have to be handled for a specific case, no code has been generated to execute a guarded action. Sequentializing the guarded actions to a list of bundles is the task that is explained in this section.
The sequentialization algorithm operates similar to the scheduler of a data flow computer, except for the number of actions that are scheduled. The bundle creation task is intended to create VLIW optimized code; therefore, the maximal parallelism provided by the program should be used. In particular, that means that not any action but all actions that can fire according to Definition 10 in Section 3.1 have to be fired. For the bundle code creation that means that all actions that can fire at the same point of time are grouped to a bundle.

Starting with an Example

Figure 4.5 shows an example for the sequentialization. The partial evaluation makes the explanation of the example more extensive, but it is not necessary to explain the sequentialization; therefore, it is neglected. Assume that a module with input variable i and the output variables x, y, z has to be synthesized. Let i, x, y, z be of the same type, either integer or natural. Furthermore, the module has some labels which do not need to be enumerated for this example. It is only necessary to make the following constraints: $\forall i = 1, \ldots, 5.\Delta(\varphi_i) \bigcap \{i, x, y, z\} = \{\}$, i. e. all variables that are listed in the module's interface must not be needed by the actions' guards. The second constraint is $\varphi_2 \wedge \varphi_3 = false \wedge \varphi_4 \wedge \varphi_5 = false$. This implies the absence of write conflicts, i. e. a variable can not be assigned more than one value in one cycle. Step one in Figure 4.5 shows the data that comes from the preceding task that rearranges the module structure. The upper node contains the known variables, which are listed in V_{known}. According to Section 3.2, V_{known} must contain all variables that are inputs, and all output variables that have no immediate action and no absence reaction. In this case, this applies to variable i, which is an input and x which has only a delayed action.
Step two builds the data flow graph by adding edges to the set of action groups. Hence, this illustrate the data dependencies between actions that read a variable and actions that can potentially change a variable. Reading a variable is always done in the current cycle; therefore, only those actions that change a variable in the current cycle are of interest when creating these dependency edges. According to Definitions 12, 13 and 31, only immediate actions and absence reactions can change a variable in the current cycle. Assignments of delayed actions

are always delayed until the start of the next cycle. Hence, the edges are directed from a source node to a destination node. The destination node is an action that needs a variable. The source node is either V_{known} if the needed variable is known according to Definition 37 or, if the needed variable has any immediate actions or an absence reaction, the action group of the needed variable. The data dependencies are always given implicitly and are drawn in this example to simplify some explanations. Reconsider, that Definition 10 can be seen as a rule for firing actions. To visualize this rule, one can imagine that an action that has no incoming edge or only input edges from the V_{known} node can fire. The actions which comply to this condition are marked by a frame with a thick dotted line. At this point, one can also see the implicitly given data independence between the actions. For example, between the action group of x and the actions of z, no edge is drawn. This means, that all actions of z are data independent of the actions in x. Although both actions of z read the value of x, the current value of x can not change, because the action group of x contains only delayed actions which can not modify the current value of x. Note that the data independence is given for actions that are marked at a step. Knowing which actions can be fired, one can generate code for these actions. This is done in Step 2b of the example. An assignment is created for each action that can be fired and all created assignments are grouped to a bundle. At this point, care has to be taken about three important points: first, an assignment is syntactically an action with the destination variable, but one has to respect the semantics difference. An assignment is assumed to consume time in contrast to an action. Second, the already mentioned data independence which is implicitly given for all actions that are fired at one step has to be considered. Therefore, their corresponding assignments can be executed at the same time or in an arbitrary order which fullfills Definition 34. Then again, they build a bundle. Finally, another kind of data dependencies has to be considered. As seen in Figure 4.5, both actions of z are fired in Step 2. The generated assignments are of immediate type, i. e. they have a potential immediate write access to z. As already mentioned in Section 3.1, one can assume that the module is free of write conflicts. Therefore, $\varphi_4 \wedge \varphi_5 = false$ is required, i. e. the guards of the actions are exclusive; therefore, the conditions of the assignments are disjunctive. Then again, this means that in one cycle at most one write access is done to z.
Now, one can proceed to Step 3a. After the creation of the assignments, the source, i. e. the actions that have been fired, have to be removed. The actions that have been marked in Step 2a are removed and so are all their incoming and outgoing edges. The action group of z does not contain anymore any immediate actions or an absence reaction, i. e. z can not be changed by an assignment anymore in the current cycle. Therefore, it is now known and, then again, z is put into V_{known}. Therefore, the start of all outgoing edges of the action group of z have to be moved to the V_{known} node. Hence, the action group of z is left empty; therefore, it can be removed. In the action group of y, only one immediate action is removed. Hence, the action group of x remains unmodified.
The creation of a bundle can be summarized as follows:

1. Mark all actions that apply the firing rule (see Definition 10)

2. Create assignments for all marked actions.

3. Remove the fired actions.
4. Remove empty action groups.
5. Check for variables that are now known according to Definition 37.

To proceed with the creation of bundle code, the next set of actions that can fire has to be determined. The start of some edges has been moved, therefore, other actions are unlocked, i. e. other actions can fire now. In particular, as already done in Step 2a, all actions are checked again if they have no incoming edge or only edges coming from V_{known}. In Step 3a, only one action fullfills the requirements. This action is marked by a thick dotted frame.
In Step 3b, an assignment is created for the marked action. Because only one action is able to fire, the created bundle contains only one assignment. Note that the new bundle is printed in black, while bundles that have been created in preceding steps are printed in gray. Having created the assignment for this action, the action itself can be removed from the graph. Because in the action group of y no immediate actions and no absence reaction left, y can not be changed anymore in the current cycle. Therefore, it is now known and added to V_{known}.
Having completed another creation of one bundle, the creation of the next bundle can be started. Working through each point of the check list, the results can be seen in Steps 4a and 4b of the graph. After the completion of the third bundle, one can see in step 5a that no action groups are left. Therefore, Step 5b is the final bundle code for the synthesis. This previous example gives an impression, how bundle code has to be created. Note that, if the partial evaluation applied, for each case a separate bundle code has to be created.

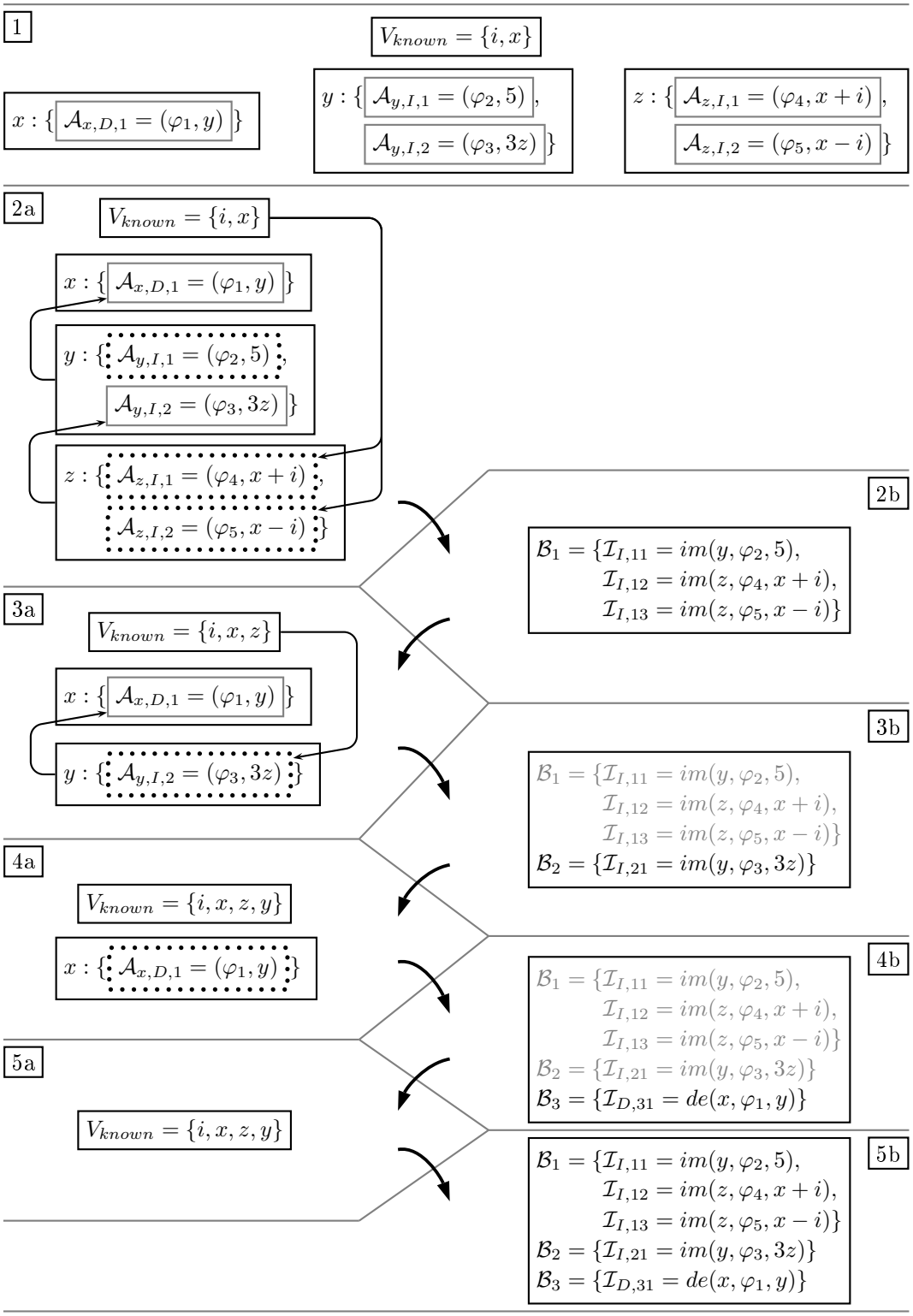


Figure 4.5: Example for sequentialization of a list of action groups.

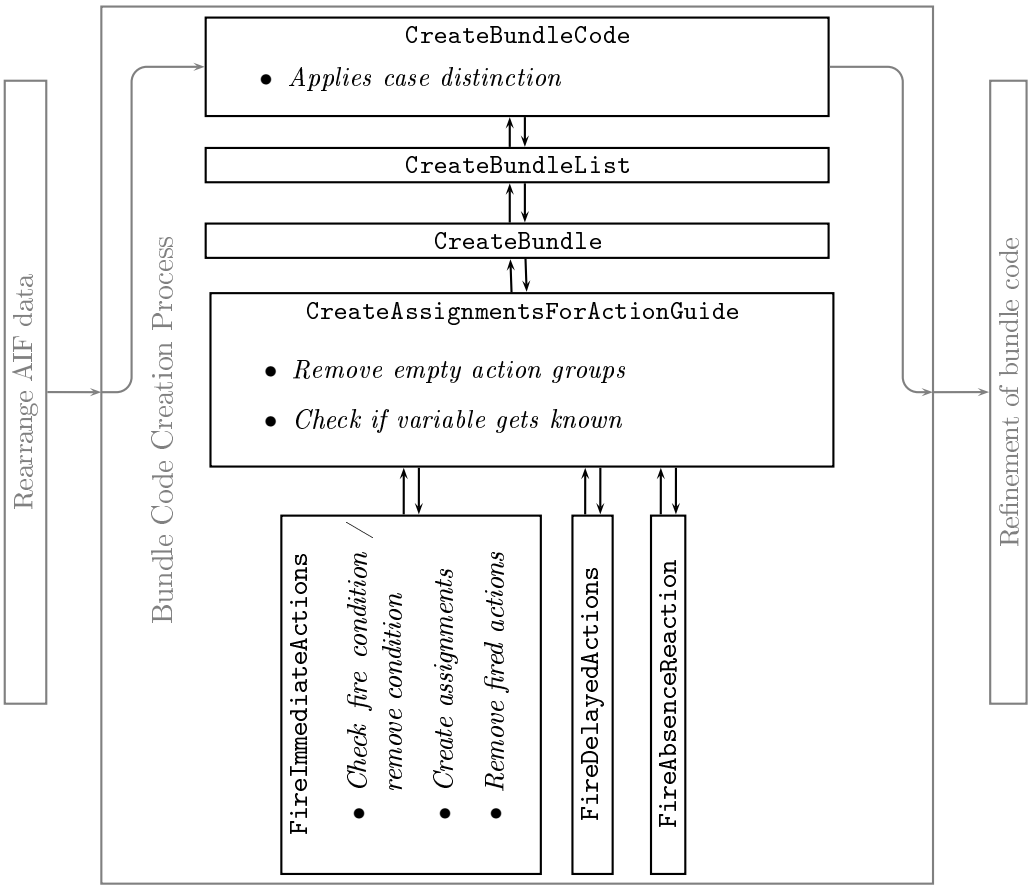


Figure 4.6: The different tasks of the bundle code creation process.

Hierarchy

Section 4.4 gave an example which anticipated a basic description of the sequentialization algorithm. Now, we are going to describe the creation of bundle code in detail. Start with an overview of the sequentialization process. As can be seen in Figure 4.6, the creation of a list of bundles is splitted into several subtasks. First, these subtasks are roughly described to get an overview of the hierarchy of the methods that are used in the algorithm. Then, these subtasks will be described in more detail. It is important to note that the resulting algorithm will not yield the optimal result for a generic VLIW processor. Therefore, the detailed description is followed by some improvements. Having an overview of the sequentialization algorithm with some details, these improvements are easier to understand. In most cases, a simple example will show where the algorithm has to be improved.
Now, consider Figure 4.6 to get a graphical overview of the algorithm's hierarchy.

- First, `CreateBundleCode` applies the partial evaluation which has already been discussed in 4.2. The partial evaluation yields a list of action groups for each case. Then, each list of action groups has to be synthesized to a list of bundles, which is done by `CreateBundleList`.

- A list of bundles is created as follows: `CreateBundleList` starts with an empty list. Iteratively, `CreateBundle` is called yielding a bundle that is appended to the list of bundles and the remaining actions that could not be fired. This is repeated until no actions are left.

- Cleary, the task of `CreateBundle` is to create a bundle. Therefore, assignments have to be created for all actions that can fire. For sake of a better structure of the implementation, this task is also splitted. The creation of assignments is separately done for each action group. This simplifies the check for new known variables which is done in `CreateAssignmentsForActionGuide` immediately after firing the actions of an action group. The results of all calls to `CreateAssignmentsForActionGuide`, i. e. the lists of assignments are concatenated to one list which builds the new bundle. The calls to `CreateAssignmentsForActionGuide` yield also a list of new variables that is also returned.

- As mentioned, the creation of a bundle is split into the creation of assignments for every action group (`CreateAssignmentsForActionGuide`). Again, this task is splitted into subtasks, i. e. the firing of the actions is done by separate methods. Firing the immediate actions and the delayed actions is done by `FireImmediateAction` and `FireDelayedAction`, respectively. Each of these functions return a list of assignments for the actions that have been fired and a list of remaining actions. In dependence of the return values, `CreateAssignmentsForActionGuide` decides whether to fire the absence reaction by `FireAbsenceReaction` or not. This task has to return all created assignments and the remaining actions. Additionally, it has to check if the variable of the processed action group has to be returned as a known variable. There are many cases that have to be checked when processing this task. They will be described in detail in Section 4.4. The subtasks `FireImmediateAction`, `FireDelayedAction` and `FireAbsenceReaction` are responsible for creating assignments from the actions. The functionality of these tasks is quite easily described: the input is a list of actions. For each action the fire condition has to be checked. If an action can be fired, an assignment has to be created for it, and the action is removed from the action list. `FireImmediateActions`, `FireDelayedActions` and `FireAbsenceReaction` return all created assignments and the remaining actions. The reason to decide between the types of actions is, `FireImmediateActions`, `FireDelayedActions` differ in the return values and `FireAbsenceReaction` takes care about some special cases that occur when firing the absence reaction. This is explained in the next section in more detail.

Detailed Description

Having an overview of the hierarchy of the sequentialization process, a more detailed description of the subtasks will be given. In contrast to the description of the hierarchy, this description will be in bottom-up order. Therefore, it starts with the explanation of `FireImmediateActions`, `FireDelayedActions` and `FireAbsenceReaction`.

Firing of Immediate Actions

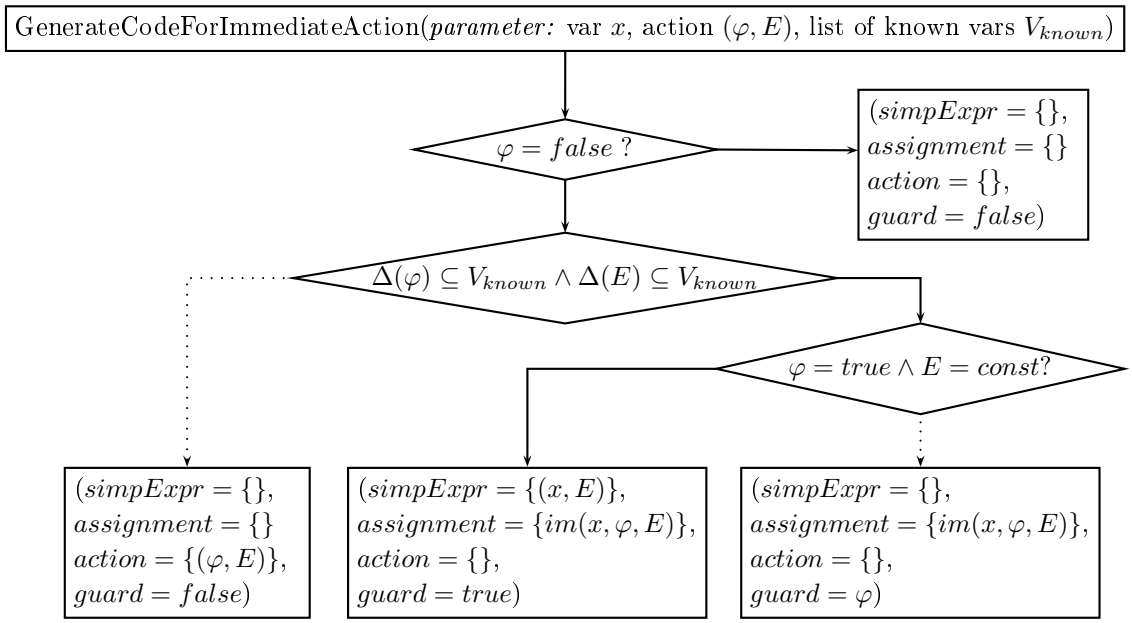


Figure 4.7: Firing a single immediate action for a variable x

`FireImmediateActions` fires a list of immediate actions. Each immediate action is processed by the function `GenerateCodeForImmediateAction`. This function has to consider different cases that can occur when firing a single immediate action. Figure 4.7 shows the decision tree for firing a single immediate action. The function `GenerateCodeForImmediateAction` has to return the following information:

- *simpExpr* (simplifying expression): A pair (x, c), where x is the destination variable and c is a constant. This pair is only returned if the assignment expression can be evaluated to a constant and the guard is true. In a later step of the firing algorithm this pair is used to replace all occurrence of this variable in all remaining actions by the corresponding constant.
- *assignment*: If the action can fire, an assignment is generated and returned.
- *action* (remaining action): If the action can not fire, it has to be returned.
- *guard*: If the action can be fired, the guard is returned. The returned guard is used to build the overall guard of all fired immediate actions.

As can be seen in Figure 4.7, it is first checked if the action is a dud, i. e. an action that can be removed, because its guard is false. If the guard is not false, the next step is to check if the action can fire, i. e. if all variables that appear in the guard and the expression are known. If the action can not be fired, it has to be returned, otherwise an assignment has to be created. Additionally, it is checked if the expression is a constant. In this case, a simplifying expression has to be created and returned.

The element-wise compound of all returned values of `GenerateCodeForImmediateAction` yields the return value of `FireImmediateActions`. In particular, the assignments are grouped to a list. The same is done with the remaining actions which yields a list of remaining actions that could not fire. The guards will be compound by a disjunction yielding an overall guard for all actions that have been fired. The semantics of the overall guard are that the destination variable will be assigned a value iff this overall guard is evaluated to true. Hence, if this overall guard can be evaluated to *true*, the evaluation of the guards of the remaining actions must be $false$. Due to the assumption that the module is free of write conflicts, no other action will be able to fire; therefore, all remaining actions can be removed.

Firing of Delayed Actions

Firing immediate and delayed actions is very similar. Firing delayed actions can be simplified because some return values are not used. To start with is the simplifying expression. Only the values of the variables in the current cycle are of interest, but delayed actions have only influence on the values in the next cycle. Therefore, the check for constant values ($\varphi = true \wedge E = const?$) and the simplifying expression can be removed.

Firing of the Absence Reaction

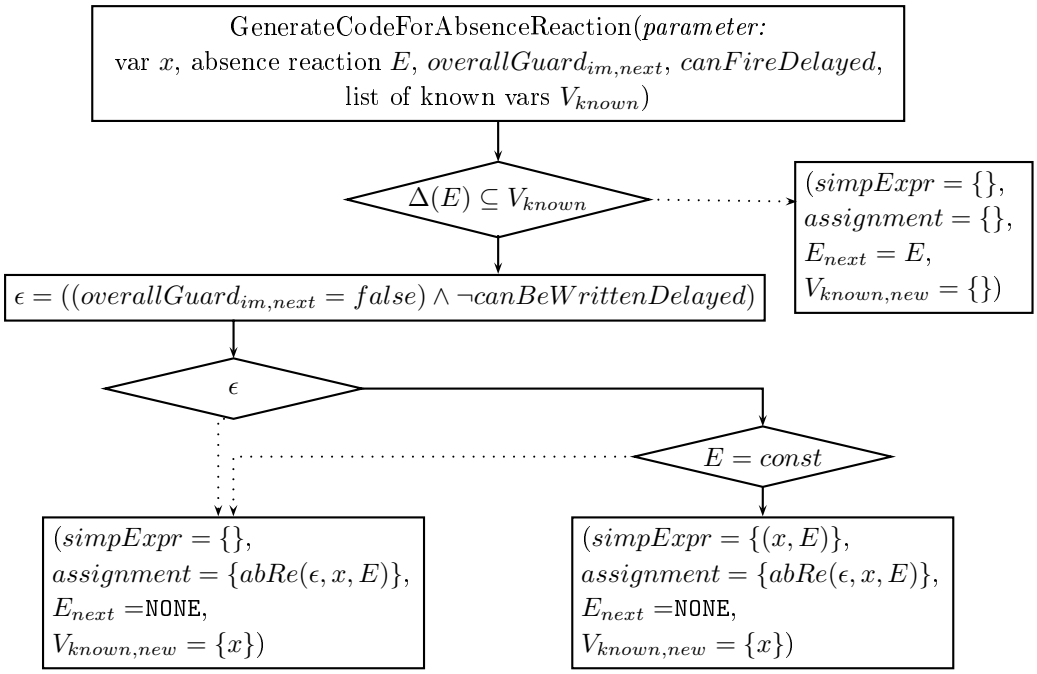


Figure 4.8: Firing the absence reaction

The absence reaction can be fired at the earliest when all immediate actions have been fired. This point of time has to be found by `CreateAssignmentsForActionGuide`, which is described in the next section. However, the ability to fire depends on more factors (see Figure 4.8), which

are handled by the subtask `FireAbsenceReaction`. The return values of this function are partially the same as the return values of the subtasks `GenerateCodeForImmediateAction`:

- *simpExpr* (simplifying expression): A pair (x, c), where x is the destination variable and c is a constant. This pair is only returned if the assignment expression can be evaluated to a constant and the guard is true. In a later step of the firing algorithm this pair is used to replace all occurrence of this variable in all remaining actions by the corresponding constant.
- *assignment*: If the action can fire, an assignment is generated and returned.
- E_{next} (remaining absence reaction): If the absence reaction can not fire, it has to be returned.
- $V_{known,new}$ (new variable): If the absence reaction can be fired, the function declares the destination variable as known.

To be able to fire the absence reaction, all necessary variables have to be known. This check is done first. Although an absence reaction has no guard, it has an implicitly defined firing condition that was described in Definition 32. The firing condition depends on the state if an immediate action has been fired in the current state or if a delayed action has been fired in the last state. While the former can be checked by the overall guard of the immediate actions, the latter can not be checked that way. The overall guard of the delayed actions determines if a delayed assignment is done in the next cycle, but not in the current cycle. The solution is given by the introduction of a state variable for each variable in the synthesized code. At the beginning of each cycle, this state variable has to be reset. The state variable has to be set as soon as an assignment is done to a variable. Hence, the firing condition can be checked at runtime. The absence reaction for a variable has to be fired iff its state variable is not set. Nevertheless, if the variable has no delayed actions at all, and all immediate actions have been rejected, because their guards were evaluated to false, the absence reaction can be fired unconditionally. In this case, the expression can be checked for a constant value. In particular, if the expression that has to be assigned is a constant, a simplifying expression can be created and returned.
Because the absence reaction is the last action of a variable that might change its value, it is at the function to declare the variable as known. This is quite simple because the variable has to be made known as soon as the absence reaction can fire.

Firing Actions of an Action Guide

`CreateAssignmentsForActionGuide` manages firing the actions for a variable x. The function gets an action guide for that variable and has to return the following values:

- *simpExpr* (simplifying expression): A pair (x, c), where x is the destination variable and c is a constant. This pair is only returned if the assignment expression can be evaluated to a constant and the guard is true. In a later step of the firing algorithm

this pair is used to replace all occurrence of this variable in all remaining actions by the corresponding constant.

- assignments: For all actions that are fired, assignments are created. These assignments build a list that is returned.

- modified action guide: The action guide that is given as an input parameter is modified. All actions that are fired are removed from the action lists. Hence, the overall guards for immediate and delayed actions may be modified if one ore more actions are fired. This has also to be considered for the absence reaction. If the absence reaction was fired, it is deleted by setting $E_{abRe,next}$ to $NONE$. However, this appears in only one case and in all other cases $E_{abRe,next}$ is implicitly set to E_{abRe}.

- $V_{known,new}$ (new variable): If all immediate actions and the absence reaction have been fired, the variable has to be declared as known. In particular, if x has to be declared as known, it has to be returned as $V_{known,new}$, otherwise $V_{known,new}$ is empty.

In general, this function has to fire the immediate and the delayed actions of variable x. Additionally, if all immediate actions have been fired, the absence reaction has to be fired according to the firing rule, i. e. if all necessary variables are known. Due to the fact that the more the generated code is optimized, the more information about a variable's value can be obtained, the goal is not only to fire the variable's actions. Care has to be taken of unconditional actions, i. e. actions whose guard is true, or the overall guard which might be true; therefore, reject the absence reaction. There are many cases that might improve the code generation.
Figure 4.9 shows a decision diagram of the process of handling the action guide of a variable. The function starts with firing the immediate and the delayed actions that are given in the action guide of x. This is done using `FireImmediateActions` and `FireDelayedActions`. Reconsider that a function does nothing, if the given list of actions is empty. Each function returns a list of assignments for all actions that have been fired. Both lists will be added to the bundle that will be created in the `CreateBundleCode` function which was already introduced in Figure 4.6. In general, both lists are concatenated and returned. If the absence reaction has to be fired, the assignment for it has to be added to the list, too.
The next part give a basic description of the further process and deepens the cases that are considered after having fired a variable's immediate and delayed actions. The firing of the absence reaction for the variable x depends on the current list of immediate actions $\mathcal{A}s_{im,current}$ of x, which is given in the action guide of x, the next list of immediate actions $\mathcal{A}s_{im,next}$ of x, which is returned by the call to `FireImmediateActions` and the overall guard $overallGuard_{im,next}$ of x of all fired immediate actions including $\mathcal{A}s_{im,current} \setminus \mathcal{A}s_{im,next}$, i. e. the actions that have been fired for the creation of the current bundle. Since the last state is unknown, one can not refer to the delayed actions that might have been fired in the previous state; therefore, have changed the value of x in the current state. Only, the general case can be considered, i. e. the state if the variable x has delayed actions is of interest.
In detail, `CreateAssignmentsForActionGuide` starts with a check if the current list of immediate actions, i. e. $\mathcal{A}s_{im,current}$ of x, is empty, `FireAbsenceReaction` can be called if an absence

reaction is given. `FireAbsenceReaction` checks further firing condtions for the absence reaction, e.g. if all necessary variables are known. `CreateAssignmentsForActionGuide` does not have to check the return values of `FireAbsenceReaction`, because they are constructed for an unmodified return. However, the potentially created absence assignment has to be added to the list of the assignments that might be created by `FireDelayedActions`. Because the list of immediate action is empty, the list of immediate assignments must also be empty. If x has no absence reaction, the given action guide contains only delayed actions. Hence, only delayed assignments might have been created; therefore, have to be returned.
If the list of immediate assignments is not empty, then one has to consider the cases that appear on the right side of Figure 4.9. If the next list of immediate assignments $\mathcal{A}s_{im,next}$ is not empty, the absence reaction can not fire; therefore, the function terminates. Otherwise, if $\mathcal{A}s_{im,next}$ is empty, one has to consider the following cases: the new overall guard $overallGuard_{im,next}$ is the disjunction of the guards of all fired actions. It might be the expression $true$, therefore, the absence reaction will not be fired, i. e. a check for firing the absence reaction is not necessary and the absence reaction can be rejected. Hence, there is no immediate action and no absence reaction that might change the value of the variable in the current cycle; therefore, the variable is known, i. e. it can be returned as a known variable. Additionally, any simplifying expression that has been created by `FireImmediateActions` has to be returned. If the overall guard $overallGuard_{im,next}$ is not the expression $true$, and variable x owns an absence reaction, the function can return the created assignments and the remaining delayed actions. The absence reaction can not fire at this point of time, because it is possible that an immediate assignment is executed in the current bundle. As mentioned, the firing of the absence reaction is triggered by an additional state variable that might be changed by the last created assignments. Therefore, the absence reaction can not be fired with the execution of the last immediate assignment, but after its execution. If x does not own an absence reaction, the function can declare the variable x as known and return the created assignment. But there is still one special case that has to be handled. If x is an event variable, i. e. an assigned or more precise an emitted value is only valid for exact one cycle. If $overallGuard_{im,next}$ is the expression $false$, then no immediate assignment will be done to x. Additionally, if x has no delayed actions, i. e. if $canBeWrittenDelayed$ is $false$, the variable x will not be assigned a value by a delayed action; therefore, it must be the events default value. In particular, if x is a Boolean variable it must be $false$, otherwise it has to be assigned 0. Hence, a simplifying expression can be created for x.

Creation of a Single Bundle

Having a function that manages the firing of actions and the extraction of information for an optimized generation of code, it is quite easy to proceed the creation of a bundle. `CreateBundle` applies `CreateAssignmentsForActionGuide` to each action guide yielding a list of assignments, a list of modified action guides, a list of variables that got known and a list of simplifying expressions. The list of assignments is returned as a bundle. The modified action guides contain the remaining actions.

Creation of a Bundle List

The next function in the hierarchy of the sequentialization process is `CreateBundleList`. This function make iterative calls to `CreateBundle` to create single bundles. This function starts with an empty list, and each new created bundle is appended to the consisting list of bundles. The simplifying expressions that are returned by `CreateBundleList` are applied to the remaining actions yielding potentially simplfied expressions and guards. The union of the returned list of new known variables and the list of already known variables yield the new list of known variables. This list is required by `CreateBundleList` as an argument. The creation of bundles is repreated until all actions have been fired. However, due to cyclic conflicts, which may appear when using the sequentialization algorithm presented so far, it is possible that not all actions can be fired. Hence, if `CreateBundleList` returns an empty bundle, the bundle list creation has to stop, too.

As already mentioned, the `CreateBundleCode` method applies the partial evaluation that has been discussed in Section 4.2. A description of its functionality has been given in Section 4.4. First, the reachable states have to be calculated. A state is in this case a configuration of a set of variables that can be assumed as known. The function creates for each configuration a separate list of bundles by using the action guides that come from the rearranging task and replace all occurrences of the variables that have a defined value in the configuration by the corresponding value. Each generated bundle list has to be bounded by an `if`-statement, whose condition is the check for the corresponding configuration of the variables.

Improvements

The sequentialization algorithm that was presented in the last section is not optimal w.r.t. the obtained parallelism. The algorithm can be improved in several points: this section will unhide the crucial points where the current algorithm is not optimal and will explain the related improvements.
The improvements that have been implemented are

- *Removal of duds:* The sequentialization algorithm is constructed to get as much information as possible from the firing of guarded actions. In some cases, a variable might be assigned a constant value, which causes the creation of a simplification expression. Applying these simplification expressions to the remaining actions can simplify a guard to $false$, i. e. the corresponding action must not fire. The separation of the removal of these actions from the firing of actions can gain a higher degree of parallelism.

- *Improving the firing rule:* This improvement begins with a modification of the definition of '*a variable is known*'. This modification takes care of the condition under which a variable is needed. This may also increase the degree of parallelism and can also break up cyclic dependencies.

- *Fixpoint iteration on simplification expressions:* Simplification expressions are used to replace variables by a constant value. Actions that depend on such a variable might

be fired earlier when the variable is replaced by a constant. Again, this improvement intends to increase the degree of parallelism.

- *Applying partial evaluation to new known variables:* Currently, the partial evaluation is only intended to be applied to variables that are known at the beginning of the synthesis process. This improvement will extend the application to variables that get known during the synthesis process.

- *Delayed Partial Evaluation:* Applying the partial evaluation duplicates the code; therefore, the synthesized program can be very large. The presented improvement delays the partial evaluation to extract common assignments yielding smaller programs. However, this technique reduces the used parallelism. Anyway, the reduction of the program's size is only interesting for large programs and large programs may provide such a large amount of parallelism that only a small amount of it can be used due to the limits of the target architecture.

- *Taking Care of Delayed Actions:* This improvement explains the correct calculation of the action guide's member *canBeWrittenDelayed* and reconsiders its usage.

Removal of Duds

This section concerns the removal of actions whose guards are $false$. In the following, these actions will be called duds. Currently, the removal of duds is done concurrently with the firing of the actions. Removing them in a separate step can improve the code generation in two ways: assume a variable that has immediate actions whose guards are $false$. The guards of these actions are not considered and, therefore, if the variable has no absence reaction, it is still unknown according to the current algorithm, because the variable still has immediate actions. Removing these duds in a separate step, the variable can be made known one bundle before. Analogously, this is valid if the variable has an absence reaction, which can be fired as soon as no immediate action is left. If the duds are removed in a separate step, the absence reaction can fire one bundle before; therefore, the variable gets known one bundle before. Making it known earlier might unlock actions of other variables, i. e. other actions might be fired one bundle earlier. Then again, this increases the average of parallelism, which is the primary goal. To implement a separate step that removes all duds is quite easy. The methods for firing the actions, i. e. `FireImmediateActions`, `FireDelayedActions` and `FireAbsenceReaction`, need a state bit that defines the firing rules that have to be used. In the following, this state bit will be called *removeDuds*. Note that only the removal of duds of the list of immediate actions will improve the synthesis, but introducing the state *removeDuds* in the `FireDelayedActions` and the `FireAbsenceReaction` is necessary to suppress the firing of delayed actions or the absence reaction. The advantage is that `CreateBundle`, `CreateAssignmentsForActionGuide` only have to forward the *removeDuds* argument to the functions that are responsible for firing the actions. Otherwise, if the state *removeDuds* is not handled by the functions `FireDelayedActions` and the `FireAbsenceReaction`, `CreateAssignmentsForActionGuide` had to decide, whether the delayed actions and the absence reaction can fire or not. Although this is possible, it

needs more programming effort and makes `CreateAssignmentsForActionGuide` more complex than `FireDelayedActions` or `FireAbsenceReaction`. Hence, `CreateBundleList` is modified so that it calls `CreateBundle` twice. The first call removes all duds by setting the state *removeDuds* to *true* yielding an empty bundle that can be ignored. The important point is the list of new known variables that have to be added to the list of known variables. Another important point is the simplification expressions that have to be applied to the remaining action guides. Reconsider that there is only one case that creates a simplification expression, when duds are removed. If no immediate action of an event variable has been fired, and the variable has neither delayed actions nor an absence reaction, then the variables value must be `false` or 0. The possibility that the removal of duds may produce simplification expression, introduces another problem. As mentioned, these simplifications are applied to the remaining action guides. Hence, the simplification of the guards of the remaining actions may result to a guard that is the expression $false$, i. e. the simplification may expose an actions as a dud. This requires to call `CreateBundle` with setting the state *removeDuds* to *true*, again. In particular, the removal of duds is done in a loop until the call to `CreateBundle` returns no simplification expressions.

Improved Firing Rule

Another improvement that might increase the average parallelism concerns the firing rule. Currently, the variables that are necessary to fire an action have to be known according to Definition 37. To modify the firing rule, this definition has to be extended to Definition 38.

Definition 38 *A variable x is known in a bundle B_l under the condition ϕ iff*
$\forall k \geq l.\neg(\exists \mathcal{I} \in B_k, \varphi, c, E.(\mathcal{I}_I = im(x, \varphi, E) \vee \mathcal{I}_{abRe} = abRe(x, c, E)) \wedge (\varphi \wedge \phi) = false)$

This allows to improve Definition 10 to Definition 39.

Definition 39 *A guarded action $\mathcal{A} = (\varphi, E)$ for the variable x has to be fired iff the variables $\Delta(\varphi)$ are known and $\Delta(E)$ are known under the condition φ, and φ can be evaluated to true.*

Reconsider that the evaluation of φ has to be done at runtime while the other condition has to be evaluated statically with the synthesis tool. This improvement of the firing rule can also break up cyclic dependencies unless they have not been broken by the partial evaluation. However, due to the current way of the translation of definitions to actions, the current solution is not correct: currently, an AIF definition (x, E) is represented by the immediate action $(true, E)$. This is sufficient if the module that has to be synthesized has no cyclic dependencies. However, consider the following example.

$$
\begin{array}{lrcl}
\mathcal{I}_{I,1} := & true & \rightarrow & def_1 = a + b \\
\mathcal{I}_{I,2} := & true & \rightarrow & def_2 = c + y \\
\mathcal{I}_{I,3} := & true & \rightarrow & def_3 = x + a \\
\mathcal{I}_{I,4} := & true & \rightarrow & def_4 = b + c \\
\mathcal{I}_{I,5} := & mode & \rightarrow & x = def_1 \\
\mathcal{I}_{I,6} := & \neg mode & \rightarrow & x = def_2 \\
\mathcal{I}_{I,7} := & mode & \rightarrow & y = def_3 \\
\mathcal{I}_{I,8} := & \neg mode & \rightarrow & y = def_4
\end{array}
$$

def_1 ... def_4 are AIF definition variables, $mode$ is a Boolean input variable, a, b and c are natural input variables and x and y are natural output variables. Furthermore, the module has the following immediate actions, where the AIF definitions have been transformed to actions according Definition 26. For a better understanding, the representation of the actions has been abstracted from the internal representation, which was explained and defined in the previous section.
Applying the sequentialization algorithm as defined so far to these action will yield the following bundles and remaining actions.

$$
\begin{array}{lrcl}
\mathcal{B}_1 := & \{def_1 = a + b, def_4 = b + c\} & & \\
\mathcal{B}_2 := & \{mode \rightarrow x = def_1, \neg mode \rightarrow y = def_4\} & & \\
\mathcal{I}_{I,2} := & true & \rightarrow & def_2 = c + y \\
\mathcal{I}_{I,3} := & true & \rightarrow & def_3 = x + a \\
\mathcal{I}_{I,6} := & \neg mode & \rightarrow & x = def_2 \\
\mathcal{I}_{I,7} := & mode & \rightarrow & y = def_3
\end{array}
$$

After the creation of second bundle, no further actions can be fired. The variables x and y are unknown under the condition $true$, therefore, the actions of def_2 and def_3 can not be fired. The actions for x and y depend on def_2 and def_3 which are also unknown. Due to this cyclic dependencies, the module can not be synthesized with the current algorithm. As one can see, the solution is quite easy. The guards of the AIF definitions have been chosen by the algorithm and are therefore not fixed. Instead of setting the guard for AIF definitions to true, it is set to the condition of its usage. In particular, if the AIF definition variable d is used in the expression E of a guarded action (φ, E), the guard of d has to be set to φ. If d appears in the expressions of several actions, the guard of d is the disjunction of the guards of these actions. Hence, if d appears in a guard of an action, the guard of d has to be set to $true$. To return to the example, the guards of the AIF definitions have to be adapted according the previous presented rule, e.g. def_1 appears only in $\mathcal{I}_{I,5}$ which has the guard $mode$, therefore, def_1 conceives the guard def_1.

$$
\begin{array}{lrcl}
\mathcal{I}_{I,1} := & mode & \rightarrow & def_1 = a + b \\
\mathcal{I}_{I,2} := & \neg mode & \rightarrow & def_2 = c + y \\
\mathcal{I}_{I,3} := & mode & \rightarrow & def_3 = x + a \\
\mathcal{I}_{I,4} := & \neg mode & \rightarrow & def_4 = b + c \\
\mathcal{I}_{I,5} := & mode & \rightarrow & x = def_1 \\
\mathcal{I}_{I,6} := & \neg mode & \rightarrow & x = def_2 \\
\mathcal{I}_{I,7} := & mode & \rightarrow & y = def_3 \\
\mathcal{I}_{I,8} := & \neg mode & \rightarrow & y = def_4
\end{array}
$$

Applying the sequentialization algorithm to the modified actions will yield the following bundles.

$$
\begin{array}{ll}
\mathcal{B}_1 := & \{mode \rightarrow def_1 = a + b, \neg mode \rightarrow def_4 = b + c\} \\
\mathcal{B}_2 := & \{mode \rightarrow x = def_1, \neg mode \rightarrow y = def_4\} \\
\mathcal{B}_3 := & \{\neg mode \rightarrow def_2 = c + y, mode \rightarrow def_3 = x + a\} \\
\mathcal{B}_4 := & \{\neg mode \rightarrow x = def_2, \neg mode \rightarrow y = def_3\}
\end{array}
$$

To complete the description of this improvement, a formal definition for the determination of the guard for an AIF definition has to be given.

Definition 40 *The action group for an AIF definition* (x, E) *is represented by the following tuple*
$(x, \{(\zeta(x, \mathcal{A}_\Sigma), E)\}, \{\}, NONE)$*, where*

- $\mathcal{A}_\Sigma$ *is the set of already existing actions.*
- $\zeta(x, \mathcal{A}_\Sigma) = \bigvee_{\mathcal{A} \in \mathcal{A}_\Sigma} a(x, \mathcal{A})$
- $a(x, \mathcal{A} := (\varphi, E)) = \begin{cases} true & , \text{ if } x \in \Delta(\varphi) \\ \varphi & , \text{ if } x \notin \Delta(\varphi) \wedge x \in \Delta(E) \\ false & , \text{ if } x \notin \Delta(\varphi) \wedge x \notin \Delta(E) \end{cases}$

Note that in general, it is not possible to assign to each AIF definition a guard variable at once. The expression that has to be assigned to an AIF definition variable may contain other AIF definition variables, i. e. the guard of an AIF definition variable a may depend on the guard of an AIF definition b. Only if the condition of b has already been calculated, the condition of a can be determined. Therefore, the action groups for the AIF definition variables have to be calculated with respect to their mutual dependencies. Due to the fact that AIF definitions can basically fire unconditional, they must not have any cyclic dependencies;therefore, the iteration for calculating the guards always has to terminate.

Exploitation of Simplification Expressions

The firing rule that creates assignments can also create simplification expressions. These simplifications can also unlock other actions which might fire simultan with the action that created the simplification expression. This can be easily explained with the following example. Given is a module with the output variables x and y and the following actions. Again, the representation of the actions and assignments is abstracted from the internal representation for a better understanding.

$$\begin{array}{|lrcl|}\hline \mathcal{I}_{I,1} := & \mathit{true} & \rightarrow & x = \mathit{true} \\ \mathcal{I}_{I,2} := & x & \rightarrow & y = 123 \\ \hline\end{array}$$

Applying the sequentialization algorithm to these actions, two bundles will be created. The algorithm starts with an empty list of known variables. The firing condition is applied to both actions yielding the result that $\mathcal{I}_{I,1}$ is able to fire, while $\mathcal{I}_{I,2}$ has to wait until x is known. Therefore, the first bundle contains an assignment that is created of $\mathcal{I}_{I,1}$. The next step is to remove $\mathcal{I}_{I,1}$ from the list of actions. Due to the unconditional assignment of x to a constant, x must be *true* in all following bundles. Hence, the algorithm creates an simplification expression, i. e. all occurrences of x in the remaining action will be replaced by *true*. Then again, the list of remaining actions is $\{\mathcal{I}_{I,2} := \mathit{true} \rightarrow y = 123\}$. The current algorithm will then create the next bundle. The action which remains does not need any variables;therefore, an assignment will be created of it. The final step is to remove $\mathcal{I}_{I,2}$ yielding an empty list of remaining actions, i. e. the sequentialization process has finished. The created bundles are shown in the following:

$$\begin{array}{|ll|}\hline \mathcal{B}_1 := & \{x = \mathit{true}\} \\ \mathcal{B}_2 := & \{y = 123\} \\ \hline\end{array}$$

As can be seen, the assignment in $\mathcal{B}_2$ is data independent of the assignment in $\mathcal{B}_1$;therefore, both assignments should have been put into a single bundle as follows:

$$\begin{array}{|ll|}\hline \mathcal{B}_1 := & \{x = \mathit{true}, y = 123\} \\ \hline\end{array}$$

This problem can be avoided by applying a fixpoint iteration to the creation of a single bundle. In particular, the call to `CreateBundle` in `CreateBundleList` has to be put into a loop. This applies to the second call of `CreateBundle` because the first call to `CreateBundle` removes the duds from the list of actions. Reconsider that `CreateBundle` returns a list of simplification expressions, that is intended to be applied to the remaining actions. Instead of applying them to the remaining actions, they have to be applied to the original list of actions. The assignments and the new known variables have to be ignored. The next step is to call `CreateBundle` again. Then again, this will return the previous list of simplification expressions and might create new simplification expressions. Therefore, the simplifications and the creation of a bundle have to be put into a loop. This loop must run until no new simplification expressions are created. The further process in `CreateBundleList` does not have to be modified. The assignments and the new known variables returned by the last call to `CreateBundle` can be processed as explained in Section 4.4.

Extending Partial Evaluation

The partial evaluation was already explained in detail. Until now, it was only intended for Boolean variables, which are known at the beginning of the synthesis process. This can be easily extended to Boolean variables that are unknown at the beginning of the synthesis. Consider the following module with the Boolean output variable x and the natural input variable y and the following actions.

$$\begin{array}{lrcl}
\mathcal{I}_{I,1} := & true & \rightarrow & c = (y \overset{?}{\geq} 99) \\
\mathcal{I}_{I,2} := & c & \rightarrow & x = 1 \\
\mathcal{I}_{I,3} := & \neg c & \rightarrow & x = 2 \\
\mathcal{I}_{I,4} := & c & \rightarrow & y = 3 \\
\mathcal{I}_{I,5} := & \neg c & \rightarrow & y = 4
\end{array}$$

The sequentialization algorithm will create the following two bundles:

$$\begin{array}{ll}
\mathcal{B}_1 := & \{c = (y \overset{?}{\geq} 99)\} \\
\mathcal{B}_2 := & \{c \rightarrow x = 1, \neg c \rightarrow x = 2, c \rightarrow y = 3, \neg c \rightarrow y = 4\}
\end{array}$$

The variable c gets known in the first bundle, i. e. it is known in the second bundle. The expression that is assigned to c can not be evaluated statically. Therefore, the value that is assigned to c can not be determined. Due to the fact that variable c is known at this point of time, a partial evaluation could be applied. This partial evaluation works exactly as described in Section 4.2. This section also explains the advantages and disadvantages of the partial evaluation. To implement this improved application of the partial evaluation, one has to modifiy `CreateAssignmentsForActionGuide` and `CreateBundleList`. `CreateAssignmentsForActionGuide` has to check if a variable can be used for partial evaluation. This decision comes with Definition 41.

Definition 41 *x is intended for partial evaluation iff x gets known and x is Boolean and x has no simplification expression.*

Additionally, `CreateAssignmentsForActionGuide` has to return a list of variables that are intended for the partial evaluation. This list has also to be returned by `CreateBundle` and `CreateBundleList`. Then again, `CreateBundleList` has to check if this list is not empty. If it is not empty, the sequentialization process has to be interrupted and the list has to be returned to `CreateBundleCode` which applies the partial evaluation to the remaining actions and then resumes the sequentialization process. However, applying the partial evaluation to more variables increases the size of the generated code.

Simplifications using BDDs

As already mentioned, some methods rewrite and simplify guards and expressions of actions. Of course, familiar logic simplifications are used to minimize formulas, e.g. $false \wedge x = false$,

$true \wedge x = x$, $false \vee x = x$, $true \wedge x = true$ and $\neg\neg x = x$. Due to the simplicity, the resulting representation of formulas may be not optimal. To improve the resulting expressions, one can convert an expression to a BDD (see [2]) and then convert it back by using different formula representations. In this diploma thesis, the BDDs are converted into DNF and the Shannon Normal Form. In general, the latter representation will result in a large formula because it creates large sub-terms. The application of BDDs makes sense when calculating guards for AIF definitions according to the improvement of the firing rule that has been explained at the beginning of the section. However, the application of BDDs to simplified expressions, respectively to a guard, will not reduce the size of expression because more complex expressions are split by the Quartz compiler into sub-expressions, whereas each sub-expression is calculated separately using AIF definitions. That means, that in general, expressions and guards will consist of only one operation; therefore they are minimized and do not have to be reduced.

Delayed Partial Evaluation

A disadvantage of the partial evaluation is the duplication of code, which yields large synthesized programs. To reduce the size of the synthesized programs, one could search for common bundles in the bundle code tree and move them up. In general, the bundles will always be at least partially different; therefore, they are not equal, and they can not be moved up in the bundle code tree. Anyway, two bundles can only be moved up if the sub paths, where the two bundles are, are connected by exact one partial evaluation. Otherwise, the program would cause the execution of instructions, which are intended for one specific sub path, in a wrong sub path. The approach that is explained in the following, gives a more confident code size reduction. However, it may reduce the used parallelism.

Currently, the partial evaluation is applied as soon as possible. This yields the minimum of guarded actions for a specific path. Since the code is duplicated every time a partial evaluation is applied, it may yield a large number of overall guarded actions; therefore, the synthesized program's size may be exponentially to the number of guarded actions and AIF definitions of the module. Instead of applying the partial evaluation as soon as possible, it may be applied after the creation of a bundle list. In particular, the order of partial evaluation and bundle code creation is swapped. All variables that are intended for partial evaluation have to be treated as unknown. The algorithm starts with creating a list of bundles. The bundles are created until the remaining guarded actions are not able to fire. If the bundle code creation left any action, the partial evaluation must be applied. This yields two groups of remaining actions, where all occurences of a variable have been replaced. After that, the algorithm is restart for each sub path. As already mentioned, the delay of the partial evaluation delays also the firing of guarded actions. Hence, the parallelism may be reduced. Another important point is the ordering of the variables. This problem occurs also in BDDs. The size of the synthesized program depends on the ordering of the variables that are intended for partial evaluation. At the moment, the ordering can not be modified. Hence, it is not possible to make any tests on this subject.
Finally, this technique is intended to be used with very large systems that provide a level of

parallelism that can not be exploited on regular VLIW processors. In particular that means that, the reduction of theoretically provided parallelism may barely influence the practically used parallelism, but the overall size of the code may be reduced.

Taking Care of Delayed Actions

Section 3.2 gave a description of the action guides. One member of this structure is *canBeWrittenDelayed*. Its intention, which was already anticipated, is the usage for optimizing the firing of absence reactions. The optimization has been explained in Section 4.4. *canBeWrittenDelayed* is used in the following two cases: if a variable can not be written by a delayed action and no immediate action was able to fire, then an absence reaction can be fired without a condition. The second case concerns the implicite absence reaction of event variables, which is handled in `CreateAssignmentsForActionGuide`. However, this requires a correct calculation of *canBeWrittenDelayed*. In all cases, *canBeWrittenDelayed* represents the state, whether in the last cycle a delayed action could have been fired or not. Therefore, one has to check the guarded actions of the last cycle to calculate *canBeWrittenDelayed*. This means that the task that rearranges the module data needs also all actions that can be fired in a cycle before. Then again, this requires the creation of three methods for simulating one cycle. One method simulates the first step, i. e. the surface, which is preceded by no actions. The second method simulates the second step, i. e. the depth that is preceded by the surface. The last method simulates all succeeding steps, i. e. the depth that is preceded by itself. In general, this increases the size of the code again. The increase of the program's size can be compensated by applying a multi pass synthesis. The first run has to check the efficiency of the synthesis of three methods, e.g. by counting the number of the mentioned cases, which use the *canBeWrittenDelayed* flag. Then a decision has to be made, whether to synthesize two or three methods, and finally, the second run applies the desired synthesis. Due to a lack of time, this improvement has not been implemented yet.

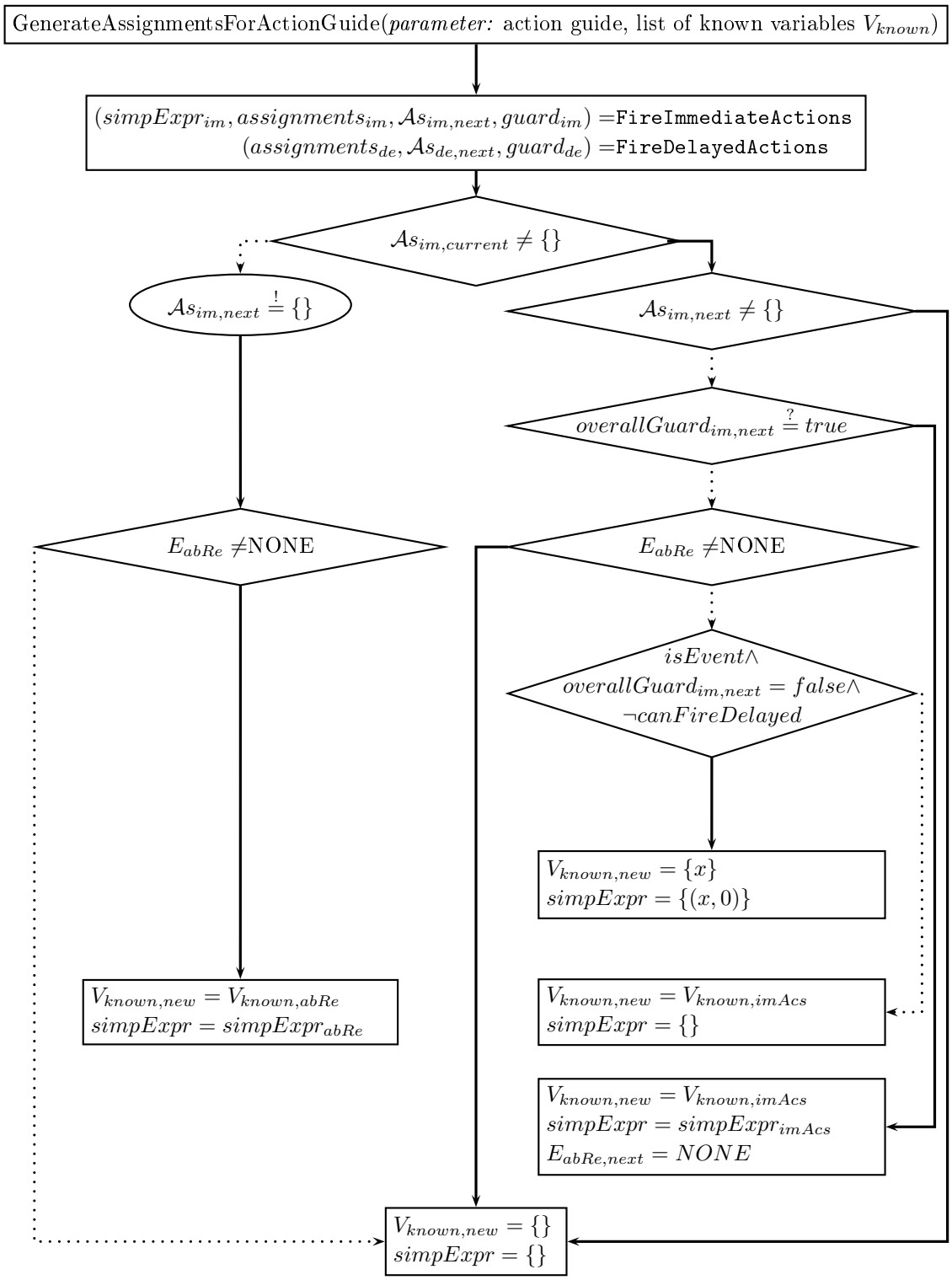


Figure 4.9: Firing actions for a variable x

5 Optimization

The task of the optimization step is to remove unused AIF definition assignments and to move them within paths to sub-paths to force the execution of these AIF definition assignments only if it is necessary. Additionally, single occurrences of an AIF definition variable can be replaced by the expression that has to be assigned to it.
One thing the synthesis process differs from original data flow graphs is the use of defined varaibles in AIF. They can be handled by creating actions that are added to the actions of the surface, and to the actions of the depth. In particular, AIF definitions are handled like local output variables. To create VLIW code for a program, an action group is created for each AIF definition. This action group contains exactly one immediate assignment, no delayed assignment and no absence reaction. The guard of the immediate action can be chosen in two ways. The first way is to assign $true$, because an AIF definition can be treated as an unguarded action. The second way determines the guard according to its appeareances (see Section 4.4, second improvement). Each method has advantages and disadvantages. The former may fire the AIF definition assignment earlier but fires it even if it is not read by any succeeding assignment. This case appears if the guards of the actions which read the AIF definition variable are simplified to $false$; therefore, this might result in non-optimal code. We know that the value of an AIF definition is only used in the same cycle of its calculation, i. e. it does not have to be calculated if it is not used in the same cycle.

Furthermore, if the partial evaluation is applied after the creation of a list of bundles $\mathcal{B}_1, \ldots, \mathcal{B}_n$ (see Section 4.4, fourth improvement), the control flow is split into two paths P_1 and P_2. If $\mathcal{B}_k, 1 \leq k \leq n$ contains an AIF definition assignment for an AIF definition variable x and x is only needed in one of the two sub paths P_1 and P_2, the AIF definition assignment for x can be moved into the sub path that needs x. This avoids the execution of the AIF definition assignment in the other sub path. Assuming a high execution probability for the latter sub path, this might give a good speed-up in the execution of the synthesized code. However, this kind of optimization introduces the following problem: if an AIF definition variable appears only in sub path P_1 of path P, and the first bundle in P_1 needs this AIF definition variable (see Figure 5.1, top), then two possiblities are given to handle this AIF definition. The AIF definition can be inserted before the first bundle in P_1 (see Figure 5.1, bottom) or appended to a bundle before that fork, that splits P in P_1 and P_2. The former solution forces the execution of the AIF definition assignment only if it is necessary. However, this solution adds an additional bundle and increases the number of steps that are necessary to execute path P_1. The second solution circumvents the problem of inserting a new bundle by moving the AIF definition assignment up to the next bundle. Note that this does not have to be the original bundle of the AIF definition assignment. As mentioned, the advantage is to avoid the creation of extra bundles but executes the AIF definition assignments also if it is not

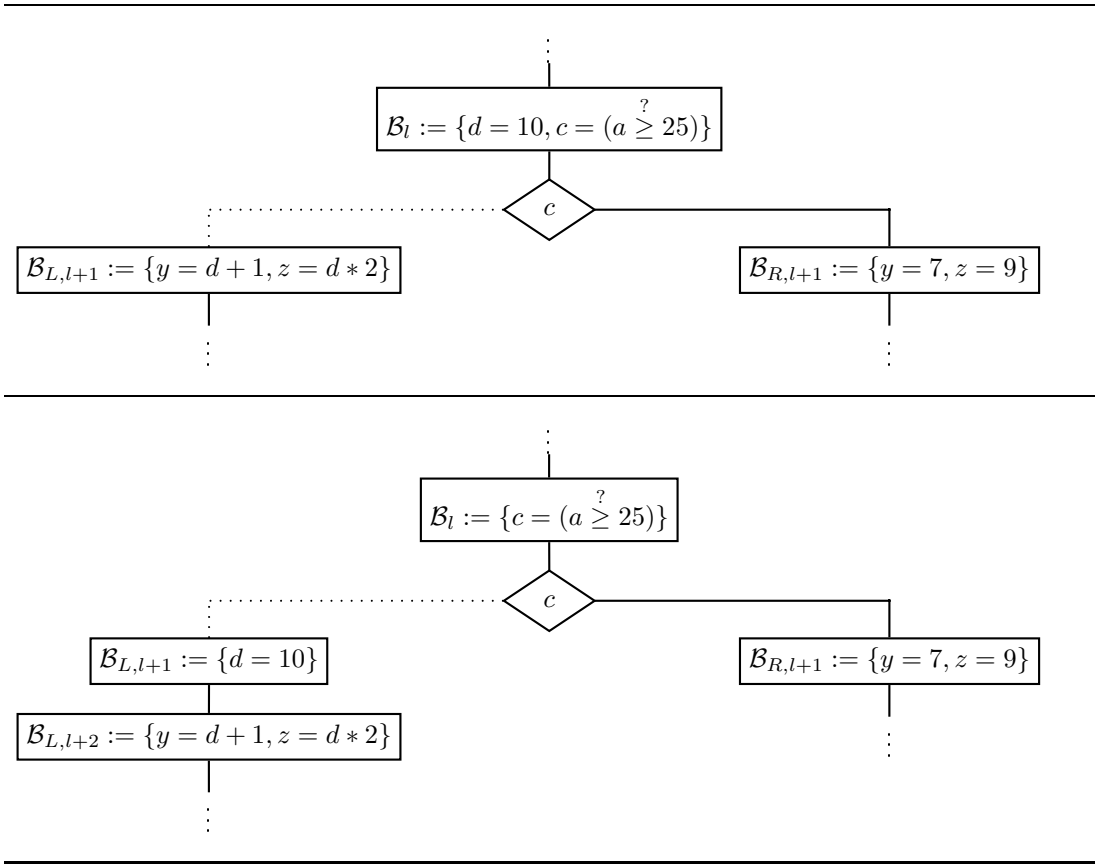


Figure 5.1: Top: An example for bundle code, Bottom: moving definition assignment $d = 10$ using *insertion mode*

necessary. Recall, that the synthesis process generates code for a generic VLIW architecture that can execute arbitrary large bundles. In practice, the benefits of the applied solution depend on the used VLIW architecture and also on the case itself. Even the knowledge of the target architecture does not imply that one of the presented solutions is always better than its alternative. In this diploma thesis, both approaches have been implemented but only one is always applied, i. e. currently, there is no smart algorithm that tries to find the best solution. Another point that offers for optimization, is the replacement of single occurrences of an AIF definition variable by the expression that has to be assigned. To avoid multiple calculations of the same expression, the AIF definitions have to be kept. However, the replacement of a single occurrence does not multiply the number of calculation of an expression but has the advantage to remove one memory write-access. VLIW processors usually have only a few memory units. If this single-occurrence-replacement rule can be applied to many AIF definition assignments, this can result in a high speed-up of the execution of the generated code.

The algorithm for the optimization of bundle code is applied recursively, starting at the bottom of the code. The optimizations that can be applied to the AIF definition assignments

in a bundle depend on the succeeding bundles; therefore, it potentially depends on other succeeding AIF definition assignments. Hence, it is a good idea to start with the optimization of any succeeding AIF definition assignments. In particular, the optimization algorithm traverses the bundle code in postorder, i. e. if a node is a bundle, the optimization is applied to the successing bundle code and then to the visited bundle itself. If the node is a fork, i. e. a partial evaluation, the optimization is applied first to all subtrees. The fork itself contains no assignments and does not have to be handled. Having defined the traversing order, the next step is to apply the optimization to a specific bundle. As described, the optimization will be applied first to the succeeding bundle code; therefore, one can assume that only the first node may contain AIF definition assignments that have to be optimized. Only if the first node is a bundle, it may contain AIF definition assignments. All AIF definition assignments have to be removed from the first bundle. The next step is to insert each AIF definition assignment that has been removed separately into the bundle code by the optimization method o. The task of the optimization method o is to move an AIF definition assignment, so that it is fired as early as possible but only if it is needed in the path where the AIF definition assignment is located. As mentioned, the optimization offers two ways to handle an AIF definition assignment whose variable is needed in the first bundle of one sub path of a fork but nowhere else.

- Method one, the *insertion mode*, creates a new bundle and inserts it before the corresponding bundle.

- Method two, the *maximal parallelism mode*, appends the AIF definition assignment to the first bundle in the first preceding path that contains bundles, maintaining a higher degree of parallelism but accepting the execution of potentially unused assignments.

Both methods work very similar but to get a better overview, each process is shown in an own graph (see Figures 5.2 and 5.3).
Method o_I uses the *insertion mode*, o_{MP} uses the *maximal parallelism mode* for the optimization of bundle code. Both methods get the AIF definition assignment Θ, the destination variable x of the AIF definition assignment and the bundle code where the AIF definition assignment has to be inserted. The bundle code is represented as a list of bundles, that can be empty, and an optional fork at the end. Hence, each of $C_1 \dots C_{n-1}$ will always be a bundle. C_n can be also a bundle, but it can also be a partial evaluation with two separate bundle codes for the sub paths. The optimization methods returns two parameters: the first parameter is the AIF definition assignment itself if it could not be inserted. Method o_I forces the insertion, therefore, the AIF definition assignment will never be returned, i. e. the return parameter is set to NONE. The second parameter is the modified bundle code if the AIF definition assignment has been inserted. Otherwise, the original bundle code is returned. Having declared the interface, it remains to define the behavior of the optimization methods.
In the following o_{MP}, i. e. the method that uses the *maximum parallelism mode* will be explained in detail. First, the method loops up the occurrences of Θ in the bundle code. Note, that not the exact number of occurrences is of interest but the occurrences in the path and sub paths. In particular, for the occurrence of x the following three cases are considered:

No occurrences in the given bundle code: If x does not appear in the list, it is not needed; therefore it does not have to be insert in the bundle code. The given bundle code is returned, i. e. the assignment is indirectly removed from the given bundle code.

Appearances on some paths in the given bundle code: This implies that C_n is a partial evaluation. The optimization has to be applied to the subtrees of the partial evaluation, i. e. o_{MP} has to be called for each sub path. However, it has to be checked if C_1 is a bundle, i. e. if $n > 1$, because in *maximum parallelism mode*, the optimization method may return the AIF definition assignment. If it is returned, it could not be added to a bundle; therefore, it has to be added to C_1. Note, that the AIF definition assignment can be appended to another bundle $C_2 \dots C_{n-1}$ but the goal is to fire the actions as early as possible.

Appearances on all paths in the given bundle code: The last case sounds trivial, but actually it can be tricky.

- If the bundle code starts with a bundle, i. e. C_1 is a bundle, one has to check if the bundle reads the AIF definition variable. o_{MP} must not create new bundles; therefore in dependence of the occurrence of x in C_1, the AIF definition assignment can be appended to C_1 or has to be returned.
- If C_1 is a partial evaluation, one has to check if x is part of the expression that describes the condition of the partial evaluation.
  - In the positive case, Θ has to be executed before the condition is evaluated; therefore it has to be returned with the unmodified bundle code.
  - The alternative case (if x does not appear in the condition of the partial evaluation) is the most interesting part. x must appear in both sub paths, i. e. the AIF definition assignment must be executed. Instead of returning it, the optimization method tries to insert a copy in each sub path. The reason for this approach can be found by considering the circumstances that cause this case. If the partial evaluation is preceeded by a bundle, the exploration of the bundle code for the occurrences of x would have been *on all paths*, too. Then again, the corresponding AIF definition assignment Θ would have been appended to the bundle and the optimization method would have been returned the modified bundle code; therefore, this case would not have been reached. This implies that the current case must be preceeded by another partial evaluation. It appears only in nested partial evaluations, and at least the first partial evaluation in this sequence of partial evaluations must have been a *x appears on some paths*-case. Returning the AIF definition assignment Θ at this point causes it to be appended to a bundle with the condition that x appears only *on some paths*. This can be avoided by trying to insert a copy of Θ into each sub path. However, this can fail, and Θ has to be returned.

Optimization o_I works very similar to that of o_{MP}. It is not necessary to repeat the explaination of the cases that can appear during the optimization. The only difference is the handling

of the AIF definition assignment Θ, if it is needed in the bundle code but can not be append to a consisting bundle. Instead of returning Θ, a new bundle containing the Θ is created and inserted at the current position, i. e. before C_1.
The optimization methods are called with the origin bundle code, starting with the bundle where Θ has been removed. Hence, if Θ can not be inserted or appended in another bundle, it will be appended to the first bundle. Due to the creation of bundle code, the first bundle must not have any data dependencies with the AIF definition variable x. Then again, the returned code will always contain the AIF definition assignment Θ unless it is not used in the bundle code at all.

As already mentioned, it is possible to replace single occurrences of x by the expression that has to be assigned to x. In this diploma thesis, this was implemented as an option. The replacement of single occurrences is executed at the beginning of the optimization methods o_I and o_{MP}. First, a check of the number of occurrences of x in the bundle code C is done. If the number of occurrences of x is not 1, the optimization proceeds as already described. If the number of occurrences of x is 1, the bundle code is searched for the occurrence of x and x is replaced by the corresponding expression. The modified bundle code is then returned by the optimization method o_I, respectively o_{MP}.
So far, the replacement of single occurrences of AIF definition variables by their expression might sound easy and uncomplicated. However, it introduces the following problem: assume a single occurrence of an AIF definition variable x that follows directly the AIF definition assignment Θ, i. e. Θ is in bundle $\mathcal{B}_j$ and x appears in an assignment $\mathcal{A}$ in bundle $\mathcal{B}_{j+1}$. The replacement of x by the corresponding expression removes a dependence of $\mathcal{A}$ to the preceding bundle. Hence, $\mathcal{A}$ has to be moved from $\mathcal{B}_{j+1}$ to the preceding bundle $\mathcal{B}_j$ if no other dependencies exist to $\mathcal{B}_j$. Note, that this movement has to be done to keep the bundle code consistent to the firing rule which implies that the actions are fired as soon as possible. Then again, the movement of the assignment $\mathcal{A}$ to the preceding bundle can cause the movement of assignments in bundle $\mathcal{B}_{j+2}$. Hence, after each movement of an assignment, it is necessary to check the assignments in the succeeding bundle for their data dependencies to the current bundle and to move them up if no dependencies are given. At this point, it is important to realize that an assignment can be moved at most to the preceding bundle. This is explained as follows: the AIF definition assignment Θ in bundle $\mathcal{B}_j$ must either have at least one data dependency to the preceding bundle $\mathcal{B}_{j-1}$ because actions are fired as soon as possible, or $\mathcal{B}_j$ has no preceding bundle or $\mathcal{B}_j$ is the first bundle after at least one partial evaluation. The latter case implies that Θ needs the value of the variable that is checked in one of the partial evaluations. Note that not the variable is needed but its value. The data dependency can be caused by the expression that has to be assigned to the AIF definition variable x or by the guard of the AIF definition assignment. In the former case, the data dependency is given to the assignment where x is replaced; therefore, the assignment can be moved at most to $\mathcal{B}_j$ which contained the assignment Θ. The latter case implies that the guard of the AIF definition assignment must be equal to the guard of the assignment $\mathcal{A}$ that reads x. The guard is created as the disjunction of the guards of the actions that read x. Additionally, the assignment $\mathcal{A}$ is

the only assignment that reads x, and this implies that either there was no other action which reads x or all other actions could not fire because their guard was evaluated to $false$; therefore the guards must be equal and must have the same dependencies to preceding assignments. Then again, this implies that the assignment $\mathcal{A}$ can not be fired earlier than the AIF definition assignment Θ.
To proceed with the explaination why assignments in succeeding bundles can only be moved up at most one bundle, reconsider again the condition that actions are fired as soon as possible. This implies that an assignment $\mathcal{A}_2$ in bundle $\mathcal{B}_j$ will always have at least one data dependency to an assignment $\mathcal{A}_1$ in bundle $\mathcal{B}_{j-1}$. Hence, if the destination variable of $\mathcal{A}_1$ is the only data dependency of $\mathcal{A}_2$ in bundle $\mathcal{B}_j$ and this data dependency is moved up on bundle, $\mathcal{A}_2$ can be moved up one bundle, too. Otherwise, $\mathcal{A}_2$ has other data dependencies and can not be moved. If $\mathcal{A}_1$ is moved up one bundle, $\mathcal{A}_2$ can moved up at most one bundle, too.

Applying these methods to all AIF definition assignments yields bundle code, that is free of unused AIF definition assignments. The remaining AIF definition assignments that are used may be replaced to reduce the computation effort in several paths. The resulting bundle code is now ready for the next step in the synthesis process - the translation to C.

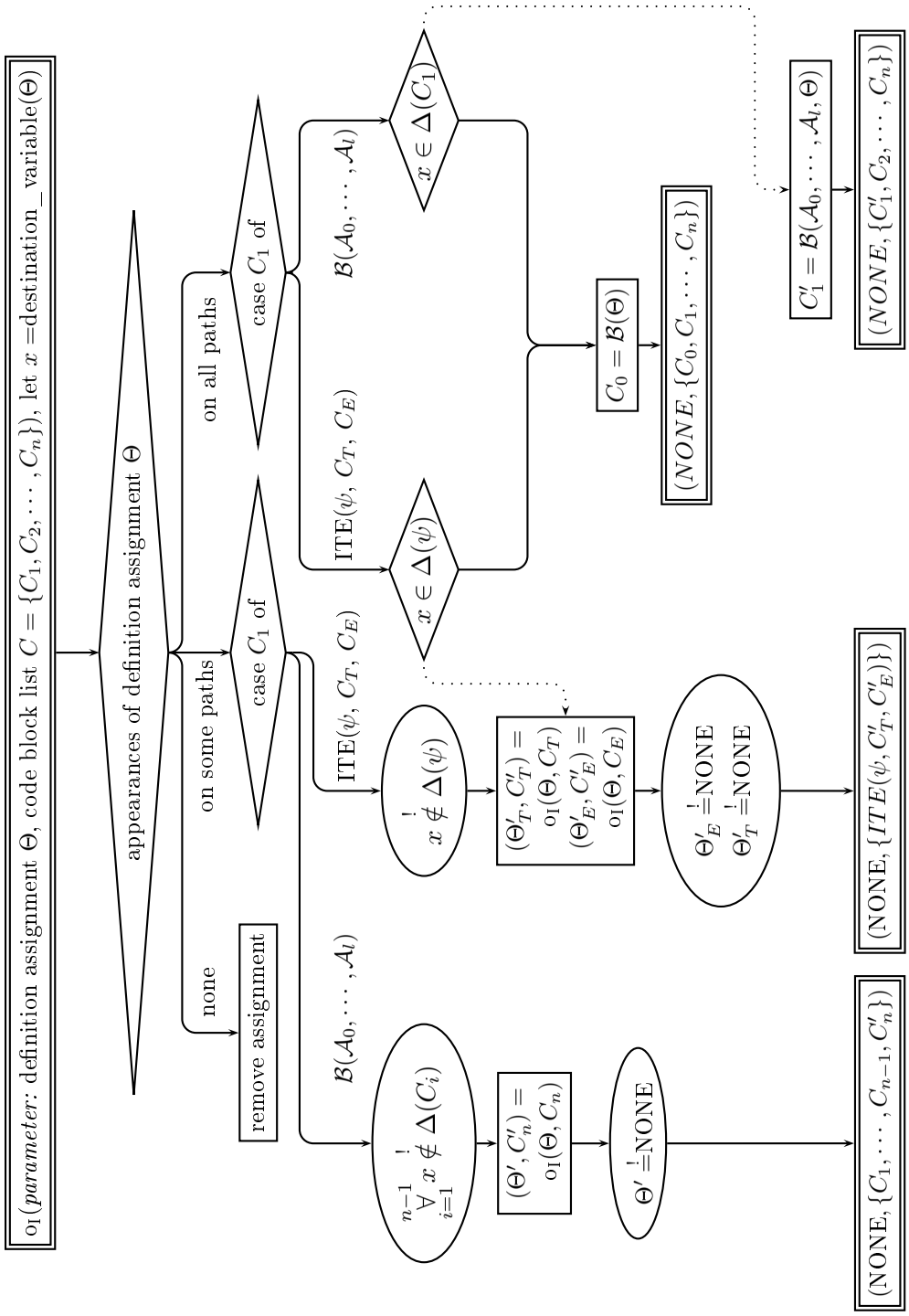


Figure 5.2: Algorithm for optimization of bundle code for a definition assignment Θ in the given code block list C using the insertion-mode.

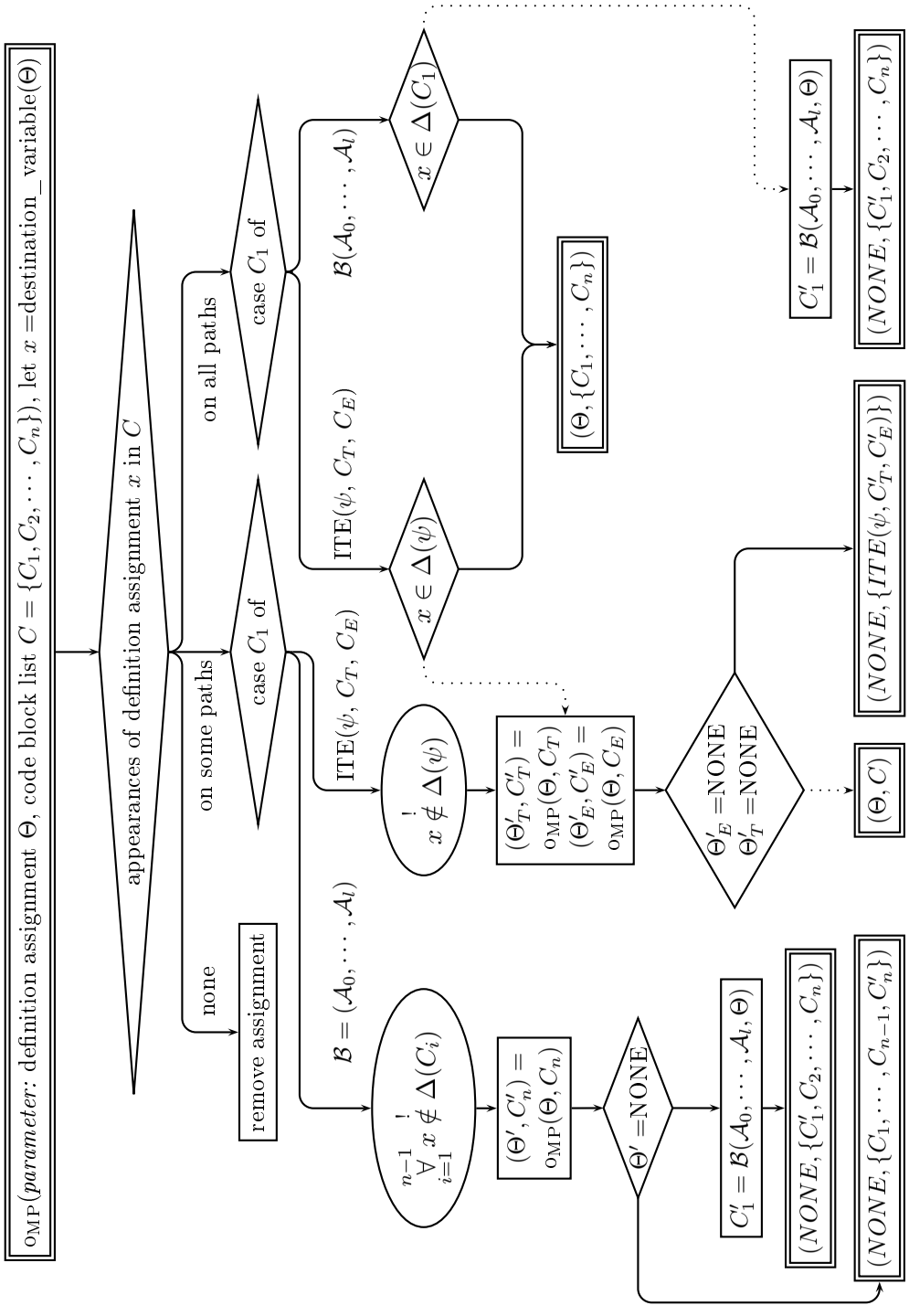


Figure 5.3: Algorithm for optimization of bundle code for a definition assignment (x, φ) in the given code block list C using the maximum-parallelism-mode.

6 Translation to C

The task described in this chapter translates the bundle code that was created by the preceding steps of the synthesis process into C code. The translation to C provides also creating declarations for all needed variables.
The translation starts with the declaration of all variables required by the module. Each variable that appears in the declaration list of the module will also appear in the declarations in the C code. For handling the different data types, the translation task is able to use several C data types. E.g. a natural number can be translated into `unsigned`, `unsigned long` or an `Natural`, which is a package for handling arbitrary large natural numbers. The C declaration of a variable also depends on the type. For each input exactly one C declaration is necessary. Outputs, local variables and labels, i. e. all writeable variables, need additional informations. A writeable variable needs a present flag, indicating if it has been written in the current cycle and a temporary storage for delayed assignments - the next value. Reconsider, that the expression of a delayed assignment has to be evaluated immediately at runtime but the result is assigned in the succeeding cycle. However, delayed assignments need some special care to avoid high computation efforts.
The basic idea for handling delayed assignments is to evaluate the expression on the right side of the assignment and to store it in the temporary storage. At the start of each cycle, a check is done if a variable contains a next value that has to be assigned. In the positive case, the value is copied and the present state is set to true, otherwise to false. However, this method requires to check each writeable variable; therefore, modules with many variables waste a lot of computations, even if no delayed assignments are done. To circumvent this problem, an array is created that logs all delayed assignments that have to be done for the next cycle. Each entry of this array contains three pointers, one for the destination, one for the source and one for the present flag, an information flag about the data type to apply the correct assignment and a flag if the destination is an event variable. An event variable has to be reset to zero after a cycle, i. e. an assignment of a value unequal to zero to an event variable causes implicitly to create a new delayed assignment of the value zero to the same variable. In this diploma thesis, the explicit triggered delayed assignments are separately executed from the implicitely delayed assignments, i. e. the reset of event variables. The reset of event variables must be done before the execution of explicitly triggered delayed assignments to avoid that a new delayed assignment is overwritten by a reset. When creating an entry in the log for delayed assignments, one can also start a search for delayed assignment to the same variable, first, but this will not be worth the effort. A search will cost more computation effort as the execution of at most two assignments.
However, it still remains to reset all present states for writeable variables. The present flag is only read by absence reaction assignments; therefore, the present flag can be omitted for

variables that do not own an absence reaction.
After the declaration, the translation task starts with the translation of the created bundle code. Reconsider that a module has a surface and a depth; therefore the sequentializing process must run two times yielding two bundle codes. The translation tasks creates two methods. The first method is an initialization method which executed the actions of the surface. It has to be called for the first cycle. For all succeeding cycles, one has to call the second method which represented the main method and executes the actions of the depth. The next step is to create method of a given bundle code.
To get a detailed description of bundle code, reconsider Section 4.1. Bundle code is a possibly empty list of bundles with an optional partial evaluation at the end. The partial evaluation refers to two bundle codes. The translation of bundle code to C is simply the translation of the assignments in the bundle to an assignment in C syntax. The partial evaluation is realized by an `if-then-else` block and two recursive calls to the C translation task with the bundle codes for the `then` and `else` path as argument. To translate an assignment to C, reconsider Definition 27 et. seq. for the assignments in Section 4. An immediate assignment $\mathcal{I}_I = im(x, \varphi, E)$ or delayed assignment $\mathcal{I}_D = de(x, \varphi, E)$ consists of a destination variable x, a condition φ and an expression E that has to be evaluated and to be assigned to x. Hence, the translation to C is `if(`φ`) x=E;` for immediate assignments, respecitvely `if(`φ`) setDelayedAssignment(x,E);` for delayed assignments where `setDelayedAssignment` is an inline function or a macro that creates an entry for the array that lists all delayed assignments that have to be done in the next cycle. An absence reaction assignment $\mathcal{I}_{abRe} = abRe(x, c, E)$ consists also of a destination variable and an expression E. But in contrast to the other assignments it has only a flag that decides wether the assignment has to be done conditionally. The condition itself is represented by the variables present flag; therefore, the translation to C is `if(!x_present) x=E;` if c is *true*, otherwise `x=E;`.
Care has to be taken of the used types, when translating expression to C code. Using the packages for arbitrary large numbers, operators have to be replaced by calls to specific functions provided by these packages.
To finish the translation task and the synthesis process, one can also create an additional method that automatically calls the initialization method at the first call and the main method for all other calls.
As one can see, the synthesis process creates no program ready for simulation. The integration of a module into a system is application-specific and has to be done manually; therefore, no `main`-method is created.
To get a better impression how the synthesized code looks like, the best is to consider an example of a synthesized module. The following example shows the synthesized module that represents a variant of the schedule problem (see Section 2.3). Some parts have been abstracted to reduce the size of the code, e.g. the loop for setting the delayed values only considers Boolean variables. The focus has been set on the functionality of the methods which simulate one cycle, i. e. `initializationSimpleSchedule`, `doMainSimpleSchedule` and `doStepSimpleSchedule`.

```
#define VARTYPE_BOOL       1

#define bool          char
#define true          1
#define false         0

// declarations
#define NUM_OUTPUT_VARS                4
#define NUM_OUTPUT_EVENT_VARS          3
#define NUM_VARS                       NUM_OUTPUT_VARS
#define NUM_EVENT_VARS                 NUM_OUTPUT_EVENT_VARS


unsigned a;       // input
unsigned b;       // input
unsigned c;       // input
bool mode;       // input
unsigned x;       // storagetype=event
unsigned y;       // storagetype=event
unsigned s;       // storagetype=event
bool ell1;       // label
bool __next__ell1;       // label
bool __present__ell1;       // label


// additional variables used to manage delayed assignments
void *DelayedAssignmentsDst[NUM_VARS + NUM_EVENT_VARS];
void *DelayedAssignmentsSrc[NUM_VARS];
bool *DelayedAssignmentsPresentState[NUM_VARS];
int DelayedAssignmentsType[NUM_VARS + NUM_EVENT_VARS];
bool DelayedAssignmentsHasToBeReset[NUM_VARS];

int whichStep = 0;
int numDelayedAssignments;
int numResetDelayedAssignments;

void setDelayedAssignment(dst, src, presentState, type, hasToBeReset) {
    DelayedAssignmentsDst[numDelayedAssignments] = dst;
    DelayedAssignmentsSrc[numDelayedAssignments] = src;
    DelayedAssignmentsPresentState[numDelayedAssignments] = presentState;
    DelayedAssignmentsType[numDelayedAssignments] = type;
```

```
    DelayedAssignmentsHasToBeReset[numDelayedAssignments] = hasToBeReset;
    numDelayedAssignments++;
}

void seta(unsigned value) { a = value; }
void setb(unsigned value) { b = value; }
void setc(unsigned value) { c = value; }
void setmode(bool value) { mode = value; }
unsigned gets() { return s; }

void initializationSimpleSchedule() {
    // definition variables
    unsigned _var1; // nat, definition (a + b)
    unsigned _var2; // nat, definition (c + y)
    unsigned _var3; // nat, definition (x + a)
    unsigned _var4; // nat, definition (b + c)
    bool _var5; // bool, definition (ell1 & mode)
    bool _var6; // bool, definition (ell1 & !mode)

    numDelayedAssignments = 0;
    numResetDelayedAssignments = 0;

    // bundle code - start
    // bundle #0
    __next__ell1 = true;
    setDelayedAssignment(&ell1, &__next__ell1, &__present__ell1,
                                        VARTYPE_BOOL, false);
    // bundle code - end
}

void doMainSimpleSchedule() {
    int currentDelayedAssignment;

    // definition variables
    unsigned _var1; // nat, definition (a + b)
    unsigned _var2; // nat, definition (c + y)
    unsigned _var3; // nat, definition (x + a)
    unsigned _var4; // nat, definition (b + c)
    bool _var5; // bool, definition (ell1 & mode)
    bool _var6; // bool, definition (ell1 & !mode)

    // delayed actions
```

```
for(currentDelayedAssignment=NUM_VARS;
      currentDelayedAssignment < NUM_VARS+numResetDelayedAssignments;
      currentDelayedAssignment++) {
   switch(DelayedAssignmentsType[currentDelayedAssignment]) {
      *((bool*)(DelayedAssignmentsDst[currentDelayedAssignment])) = false;
      break;
   }
}
numResetDelayedAssignments = 0;
for(currentDelayedAssignment=0;
      currentDelayedAssignment < numDelayedAssignments;
      currentDelayedAssignment++) {
   switch(DelayedAssignmentsType[currentDelayedAssignment]) {
      *((bool*)DelayedAssignmentsDst[currentDelayedAssignment]) =
         *((bool*)DelayedAssignmentsSrc[currentDelayedAssignment]);
      if(DelayedAssignmentsHasToBeReset[currentDelayedAssignment] &&
         *((bool*)(DelayedAssignmentsSrc[currentDelayedAssignment]))) {
         DelayedAssignmentsDst[NUM_VARS+numResetDelayedAssignments] =
            DelayedAssignmentsDst[currentDelayedAssignment];
         DelayedAssignmentsType[NUM_VARS+numResetDelayedAssignments] =
            DelayedAssignmentsType[currentDelayedAssignment];
         numResetDelayedAssignments++;
      }
   }
   if(DelayedAssignmentsPresentState[numDelayedAssignments]!=0)
      *((bool*)DelayedAssignmentsPresentState[numDelayedAssignments]) = true;
}
numDelayedAssignments = 0;

// bundle code - start
if (mode) {
   if (ell1) {
      // bundle #0
      __next__ell1 = true;
      setDelayedAssignment(&ell1, &__next__ell1, &__present__ell1,
                                          VARTYPE_BOOL, false);
      x = (a + b);
      // bundle #1
      s = x;
      // bundle #2
      y = (x + a);
   }
```

```
    } else {
        if (ell1) {
            // bundle #0
            __next__ell1 = true;
            setDelayedAssignment(&ell1, &__next__ell1, &__present__ell1,
                                                VARTYPE_BOOL, false);
            y = (b + c);
            // bundle #1
            s = y;
            // bundle #2
            x = (c + y);
        }
    }
    // bundle code - end
    // no present states to reset
}

void doStepSimpleSchedule() {
    if(whichStep==0) {
        whichStep=1;
        initializationSimpleSchedule();
    } else {
        doMainSimpleSchedule();
    }
}

// EOF
```

7 Results

7.1 Synthesis Benchmarks

The quality of the synthesized programs was measured with some hand-made AIF modules. Table 7.1 shows the examples and some numbers to get a weak estimation of the size of the examples. The numbers of writable variables exclude the state variables which are listed separately. Furthermore, the table lists the number of actions of the data flow in the depth. The number of actions in the surface are omitted because the direct influence of the surface complexity considers only one cycle. The last column gives the number of definitions that are used in the corresponding modules. Due to the creation of actions for definitions, this is also of interest.
The first matrix multiplication reads two matrices element-wise and multiplies them in one cycle. This is not usual for processors but many vector processors can apply several multiplication-and-addition at once. Furthermore, the benchmark of this example represents modules that provide a high level of parallelism. The second matrix multiplication can be interpreted as representant for an economical processors. This example applies the multiplication element-wise, i. e. an element of the target matrix is calculated in each cycle. This saves gates and reduces the size of the chip. The simple schedule is the Quartz implementation of the cyclic example shown in Section 2.3. It checks the functionality of the improved firing rule and the partial evaluation. Each of these techniques must be able to break up the cyclic dependencies. The last two modules generate a sequence of square number and the Fibonacci numbers, respectively. Each starts with the first element in its sequence, i. e. 0, and outputs one element in each cycle.
Figure 7.2 shows the results of the synthesis of the presented examples. The examples were

program	wv	sv	a	def
3x3 matrix multiplication 1-cycle	5	3	12	5
3x3 matrix multiplication multi-cycle	5	3	4	5
simple schedule	3	1	6	6
squares	2	1	2	4
fibonacci	2	1	2	1

Figure 7.1: The AIF examples for the benchmark. **(wv)** number of writeable variables; **(sv)** number of state variables; **(a)** number of actions; **(def)** number of definitions

synthesized using three different configurations. The first configuration just sequentializes the given module without applying the partial evaluation. A goal of the benchmarks is to show

program	sm	la	av	shp	lop	avp	appe
3x3 matrix multiplication 1-cycle$^{(1)}$	1	10	4.7	3	3	3.0	0
3x3 matrix multiplication 1-cycle$^{(2)}$	1	10	1.6	1	3	1.8	4
3x3 matrix multiplication multi-cycle$^{(1)}$	1	3	2.0	3	3	3.0	0
3x3 matrix multiplication multi-cycle$^{(2)}$	1	3	1.2	1	3	1.8	4
simple schedule$^{(1)}$	2	3	2.2	4	4	4.0	0
simple schedule$^{(2)}$	1	1	1.0	4	4	4.0	2
squares$^{(1)}$	1	1	1.0	3	3	3.0	0
squares$^{(2)}$	1	1	1.0	3	3	3.0	1
fibonacci$^{(1)}$	1	2	1.5	2	2	2.0	0
fibonacci$^{(2)}$	1	2	1.5	2	2	2.0	1

Figure 7.2: Technical data of the synthesized AIF examples: **(sm)** number of assignments in smallest bundle; **(la)** number of assignments in largest bundle; **(av)** average number of assignments in bundles; **(shp)** number of bundles on shortest path; **(lop)** number of bundles on longest path; **(avp)** average number of bundles on paths; **(appe)** applied evaluations; ($^{(1)}$) no partial evaluation; ($^{(2)}$) partial evaluation

the efficiency of the partial evaluations. Configuration (2) applies the partial evaluation only to the variables, that are known right from the start. The values shown in this table have been generated by the synthesis tool. The first three columns give some information about the degree of parallelism, i. e. the minimal number of assignments in a bundle, the maximum and the average. It is important to know that the minimum and the maximum values are only peak levels. Reconsider that the code creation intends to execute an assignment as early as possible; therefore, it may be possible to move assignments without violating data dependencies and without creating new bundles, so that the number of assignments in the bundles are smoothed and the peaks reduced. The average number of assignments in a bundle is calculated as follows: the average size of the bundles are calculated for each path. The overall average, that is shown in this table, is calculated as the average of the bundles' average sizes. Note, that the assignments in a bundle can not be compared to assembler instructions. In general, the assignments consists of several assembler instructions; therefore, the bundle for a generic VLIW processor may be larger than the bundle that was created by the synthesis tool. The next three columns give some information about the length of the paths. The number of bundles, that are executed on the shortest path, the longest path and the average number of bundles, that have to be executed. Note, that these information do not depend on probabilities, i. e. all paths are assumed to have the same probabilities to be executed. The last column lists the number of partial evaluations that have been applied to the modules.
As can be seen, the single-cycle matrix multiplication provides the highest amount of parallelism. That was to be expected because it executes a lot of data independent operations. The matrix multiplication module has three Boolean control signals for the combinational part of the module and one control label; therefore, four partial evaluations can be applied to it. Furthermore, the partial evaluation reduces the average size of the bundles dramatically

by factor three. However, the peak remains, which is the multiplication of the matrices itself. The number of bundles, that have to be executed is also reduced. The average has been reduced by one bundle which ends in a percentual speed-up of 33 percent.
The multi-cycle multiplication is a reduced version of the single-cycle multiplication. It calculates only one element per cyclce; therefore, it does not support the same high amount of parallelism as the single-cycle multiplication. Nevertheless, one can see the same effects as in the last example. The application of the partial evaluation reduces the average size of the bundles to less than the half. The average number of bundles, that have to be executed is also reduced by the same amount as in the last example.
The result of the synthesis of simple schedule was to be expected. The partial evaluation reduces the number of actions that have to be sequentialized to the half in each path; therefore, each path can run with one assignment per bundle. The average number of bundles, that have to be executed does not change. This was also to be expected because the order of the assignments can not be changed.
Due to the small size of the last two examples, they can not benefit from the partial evaluation or the extracting of parallelism.
The next point that has to be discussed is the synthesis time for the modules. In general, all modules were synthesized within one second. However, the matrix multiplication took about five seconds on a Pentium IV with 2.8 GHz.

7.2 Runtime Benchmarks

Figure 7.3 shows the benchmark results. These benchmarks were made using the VEX system [13] provided by HP Invent. The VEX system is designed to compile and simulate programs for VLIW architectures and is capable of using and simulating several VLIW architectures. Due to the interest in a result that is not bounded to any limits of a specific VLIW architecture, an own model of a VLIW architecture has been created. It provides an issue width of 32 instructions and the processor consists of 32 ALUs, multipliers, memory units and of 256 registers. Additionally, all programs were run with the RISC model that comes with the VEX system. The VEX Compiler provides options for setting the degree of optimization. The optimization level was set to maximum but the level of loop unrolling was set to zero to measure only the IPC of non-overlapped code. All programs have been compiled within one second on a Pentium IV with 2.8GHz.
For each type of matrix multiplication, two benchmarks were considered. The matrix multiplications that are signed with ‘wi’ apply the initialization, which is done element-wise. Therefore, they do not provide much parallelism. To get an idea of what can be reached with a highly parallelized program the second benchmark of the matrix multiplication (signed with ‘hp’) omits the initialization and executes only multiplications. However, as can be seen in Figure 7.3 this increases the IPC only slightly. The solution shows the assembler code, created by the VLIW compiler. Although, the assignments for calculating the matrix values can be interleaved, the compiler calculates each value separately. The calculation of one matrix element does not provide much parallelism; therefore, the overall IPC is not increased as it

was to be expected.
Additionally, the VEX Compiler seems to avoid problems that may be caused by invoking methods, especially if pointer are given as arguments. In the synthesized program of the 1-cycle matrix multiplication, it is possible to change the order of the calls to `setDelayedAssignment`, which initializes a delayed value assignment (see Section 6). The analysis of the assembler file, which is created by the VEX Compiler, indicated that a call to this method forces that instructions that appear before the call are executed before it, and instructions that appear after the call are executed after it. This order must be kept for the instructions of one delayed assignment, but the instructions of several delayed assignments may be interleaved. Consider Figure 7.4 for an example. The left side shows the code of two delayed assignments. Each assignment consists of two instructions. According to the implementation of `setDelayedAssignment` as shown in Section 6, the first call to `setDelayedAssignment` is independent of the second instruction; therefore, the order can be changed as shown on the right side of Figure 7.4. Hence, the two assignments can be executed in parallel. However, this independence is not recognized by the VEX Compiler. One can suppose that the compiler forces the given order due to the usage of pointer. To circumvent this problem, all methods have to be declared as `inline` methods, or they have to be replaced by macro definitions. Benchmarks with synthesized modules that were manually rewritten have shown that the IPC still remains small if the methods are replaced by macro definitions. However, the number of execution cycles can be reduced drastically, e.g. for the matrix multiplication the number was reduced to a quarter.
As can be seen, the IPCs of the VLIW benchmarks are higher than the RISC benchmarks. This was to be expected. However, the IPCs for the VLIW benchmarks are usually small, even for the 1-cycle matrix multiplication. Although, the 1-cycle matrix multiplication provides a high amount of parallelism, the main loop in the framework contains many calls for communicating with the module's interface, i. e. setting the module's input variables and reading the output variables, respectively. As already mentioned, the VEX Compiler avoids problems that may be caused by the invocation of methods by executing the invocations in order.
To measure the efficiency of the partial evaluation, each module was synthesized additionally without partial evalation (marked with $(^{(1)})$). As can be seen, the representation of the synthesized code using partial evaluation forces the compiler to use branches instead of predicated execution. In general, the IPC is reduced but the needed execution cycles and the simulated execution time are also reduced. In the most cases, a speed up of about 10 percent can be achieved. It still remains to examine the efficiency of the application of partial evaluation to larger modules.
Katz shows in [14] some results of benchmarks on VLIW processors in dependence of the branch prediction. These results demonstrate the necessity of a good branch prediction. However, the branch prediction of the VEX system can not be changed, e. g. to perfect or no prediction. This option would have been interesting, especially due to the relatively high number of branches compared to the number of other instructions.

7.3 Conclusion

Due to the small size of the presented examples, the synthesis tool was only able to generate code that provides a small amount of parallelism. However, this small amount of parallelism has been used by the VEX System and the results give some first impressions.
A positive result gave the application of partial evaluation. The partial evaluation can achieve a speed up about 10 percent, even for small examples. The application of the partial evaluation in the synthesis of synchronous programs is also able to break up cyclic dependencies and to create efficient code at the same time. The examination of the code size did not bring any significant results. In fact, the size of the effective code was always smaller than the size of the framework. However, two points remain to be examined: first, the efficiency of the application of partial evaluation to larger modules, and second, the size of synthesized code of larger modules.
Due to the functionality of the modules, they only contain Boolean variables that are input variables or output variables that are only written by delayed actions. Therefore, the extended partial evaluation, which applies the partial evaluation to new known variables, could not be applied. Another technique, whose efficiency remains to be examined, is the delayed partial evaluation. It was applied to the modules but due to their small size, the results are not significant enough to make any conclusions.
Due to the overhead that is produced by the framework of the synthesized program and the simulation program, one can assume that the theoretically provided parallelism and the practically used parallelism coinside only for very large modules.
Although, the synthesis is intended to be used for VLIW architectures, the results show that the application of partial evaluation improves the code also for sequential processors. Furthermore, the separation of the translation task and the bundle code creation task gives the opportunity to extend the synthesis tool to translate the AIF modules easily to other emperial languages, e.g. Java.
Some modifications in the translation task gave a reduction of the execution cycles and of the execution time. Although the reduction of execution cycles is a nice result, the goal that complies with the title of this thesis is to increase the IPC.
An advantage of the partial evaluation, which was not mentioned yet, is the usage for optimizations in the development process of the module. One can add counters in the sub paths to create statistics and probabilities for the sub paths. This can help to optimize computing expensive calculations on specific paths, e.g. those with the highest probability. This may be of interest in the development of software that is intended to have a minimal overall execution time. Furthermore, the synthesis tool already gives some information about the synthesized program, e. g. the number of bundles on the longest path. This could be extended to output the variable configuration, i. e. the state. Then, this could be used to optimize the worst case reaction time, which is of interest in embedded systems with hard realtime requirements.

program	model	overall cycles (execution time/msec)	execution cycles	IPC (no stalls)	IPC (with stalls)
mat. mult. 1-cycle, wi (1)	risc	29366 (0.058732)	20831 (70.94%)	0.90	0.64
mat. mult. 1-cycle, wi (1)	vliw	21778 (0.043556)	12879 (59.14%)	1.52	0.90
mat. mult. 1-cycle, wi (2)	risc	22352 (0.044704)	14156 (63.33%)	0.85	0.54
mat. mult. 1-cycle, wi (2)	vliw	19137 (0.038274)	10535 (55.05%)	1.15	0.64
mat. mult. 1-cycle, hp (1)	risc	64928 (0.129856)	57399 (88.40%)	0.88	0.78
mat. mult. 1-cycle, hp (1)	vliw	42347 (0.084694)	34073 (80.46%)	1.51	1.22
mat. mult. 1-cycle, hp (2)	risc	58020 (0.11604)	50841 (87.63%)	0.86	0.75
mat. mult. 1-cycle, hp (2)	vliw	39630 (0.07926)	31751 (80.12%)	1.39	1.11
mat. mult. multi-cycle, wi (1)	risc	24215 (0.04843)	17124 (70.72%)	0.86	0.61
mat. mult. multi-cycle, wi (1)	vliw	19417 (0.038834)	12088 (62.25%)	1.18	0.73
mat. mult. multi-cycle, wi (2)	risc	23320 (0.04664)	15693 (67.29%)	0.85	0.57
mat. mult. multi-cycle, wi (2)	vliw	19854 (0.039708)	11694 (58.90%)	1.15	0.68
mat. mult. multi-cycle, hp (1)	risc	20431 (0.040862)	16494 (80.73%)	0.84	0.68
mat. mult. multi-cycle, hp (1)	vliw	15824 (0.031648)	11378 (71.90%)	1.26	0.90
mat. mult. multi-cycle, hp (2)	risc	19741 (0.039482)	15704 (79.55%)	0.83	0.66
mat. mult. multi-cycle, hp (2)	vliw	15411 (0.030822)	10955 (71.09%)	1.23	0.88
simple schedule (1)	risc	263805 (0.52761)	233929 (88.67%)	1.03	0.91
simple schedule (1)	vliw	215573 (0.431146)	184941 (85.79%)	1.32	1.13
simple schedule (2)	risc	229461 (0.458922)	201961 (88.02%)	1.02	0.90
simple schedule (2)	vliw	187874 (0.375748)	159300 (84.79%)	1.33	1.13
squares (1)	risc	311781 (0.623562)	285823 (91.67%)	0.82	0.75
squares (1)	vliw	235000 (0.47)	205865 (87.60%)	1.17	1.03
squares (2)	risc	283719 (0.567438)	257851 (90.88%)	0.82	0.75
squares (2)	vliw	215974 (0.431948)	186884 (86.53%)	1.17	1.01
fibonacci (1)	risc	310781 (0.621562)	284823 (91.65%)	0.82	0.76
fibonacci (1)	vliw	233002 (0.466004)	203867 (87.50%)	1.19	1.04
fibonacci (2)	risc	282719 (0.565438)	256851 (90.85%)	0.82	0.75
fibonacci (2)	vliw	215929 (0.431858)	186884 (86.55%)	1.17	1.01

Figure 7.3: Benchmarks with the VEX compiler. **(1)** no partial evaluation; **(2)** partial evaluation; only matrix multiplications: **wi** with initialization; **hp** only multiplications

```
__next__C = X+Y;
setDelayedAssignment(&C, &__next__C, &__present__C, VARTYPE_INT, false);
__next__D = X*Y;
setDelayedAssignment(&D, &__next__D, &__present__D, VARTYPE_INT, false);
```

```
__next__C = X+Y;
__next__D = X*Y;
setDelayedAssignment(&C, &__next__C, &__present__C, VARTYPE_INT, false);
setDelayedAssignment(&D, &__next__D, &__present__D, VARTYPE_INT, false);
```

Figure 7.4: Top: an example for two delayed assignments created by the synthesis tool; bottom: a valid modification that increases the IPC - the first two instructions can be executed concurrently

Bibliography

[1] *The Esterel v7 Reference Manual Version v7 10 for Esterel Studio 5.0*, May 2003.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transaction on Computers C-27*, pages 509–516, 1976.

[3] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Horsholm, Denmark, March 1992.

[4] D. Baudisch. Implementierung und Verifikation eines RISC-Prozessors in Averest. Master's thesis, University of Kaiserslautern, Department of Computer Science, 2006.

[5] G. Berry. The Esterel v5 language primer. http://www-sop.inria.fr/esterel.org/, April 1997.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[7] Massimiliano Chiodo, Paolo Guisto, Attila Jurecska, Luciano Lavagno, Ellen Sentovich, Harry Hsieh, Kei Suzuki, and Alberto Sangiovanni-Vincentelli. Synthesis of software programs for embedded control application. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 587–592, New York, NY, USA, 1995. ACM.

[8] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[9] S. Edwards. Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):141–187, 2003.

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

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528, pages 1–13207–218. Springer Verlag, 1991.

[13] HP Invent. VEX Toolchain, 2005. URL: `http://www.hpl.hp.com/downloads/vex/`.

[14] Randy H. Katz. Lecture 13: Trace scheduling, conditional execution, speculation, limits of ilp. Computer Science 252, 1996.

[15] Tobias Schuele Klaus Schneider, Jens Brandt. 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.

[16] X. Li, M. Boldt, and R. von Hanxleden. Compiling Esterel for a multi-threaded reactive processor. Technical Report 0603, Christian-Albrechts-Universität Kiel, Department of Computer Science, 2006.

[17] X. Li, M. Boldt, and R. von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 303–314, San Jose, USA, 2006. ACM.

[18] S. Malik. Analysis of cyclic combinational circuits. In *International Conference on Computer Aided Design (ICCAD)*, pages 618–625, Santa Clara, CA, USA, 1993. IEEE Computer Society.

[19] P. Sestoft N.D. Jones, C.K. Gomard. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[20] Dionisios N. Pnevmatikatos and Gurindar S. Sohi. Guarded Execution and Branch Prediction in Dynamic Instruction Level Parallel Processors. Technical Report TR 1193, 1993.

[21] K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[22] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, 2008.

[23] K. Schneider, J. Brandt, and E. Vecchie. Efficient code generation from synchronous programs. In 165-174, editor, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 165–174, Napa Valley, California, 2006. IEEE Computer Society.

[24] K. Schneider, J. Brandt, and E. Vecchie. Modular compilation of synchronous programs. In *IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*, Braga, Portugal, 2006. Springer.

[25] K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems. In *Conference on Application of Concurrency to System Design (ACSD)*, St. Malo, France, 2005. Participant's proceedings.

[26] Klaus Schneider. System description languages. Presentation of Course, 2005.

[27] Klaus Schneider. Verification of reactive systems. Presentation of Course, 2005.

[28] Klaus Schneider. Parallel computers. Presentation of Course, 2007.

[29] Ashutosh Suresh Trivedi. *Techniques in Symbolic Model Checking.* PhD thesis, Indian Institute of Technology, 2003.

[30] D.W. Wall. Limits of instruction-level parallelism. In *International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188, Santa Clara, California, United States, 1991. ACM Press.

[31] M. Wenz. Codeerzeugung fuer die synchrone Modellierungssprache Quartz. Master's thesis, University of Karlsruhe, Institute for Computer Design and Fault Tolerance, 2001.

[32] J. Zeng, C. Soviani, and S.A. Edwards. Generating fast code from concurrent program dependence graphs. In D. Whalley and R. Cytron, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 175–181, Washington, DC, USA, 2004. ACM.