Interleaving Behavioral and Cycle-Accurate Descriptions for Reconfigurable Hardware Compilation

José Gabriel F. Coutinho, Jun Jiang and Wayne Luk
Department of Computing, Imperial College London
{jgfc,jj2,wl}@doc.ic.ac.uk

Abstract

This paper describes Haydn, a hardware compilation approach which aims to combine the benefits of cycle accurate descriptions such as ease of control and performance, and the rapid development and design exploration facilities in behavioral synthesis tools. Our approach supports two main features: deriving architectures that meet performance goals involving metrics such as resource usage and execution time, and inferring design behavior by generating behavioral code that is easy to verify and modify from scheduled designs such as pipeline architectures. We report four recent developments that significantly enhance the Haydn approach: (a) a design methodology that supports both cycle-accurate and behavioral levels, in which developers can move from one level to the other; (b) an extended scheduling algorithm which supports operation chaining, pipelined resources (with different latencies and initiation intervals), forwarding technique for loop-carried dependencies, and resource sharing and control; (c) a hardware design flow that can be customized with a script language and extended simulation capabilities for the RC2000 board; and (d) an evaluation of our approach using various case studies, including 3D free-form deformation (FFD), Gouraud shading, Fibonacci series, Montgomery multiplication, and one-dimensional DCT. For instance, our approach has been used to produce various FFD designs in hardware automatically; the smallest at 137MHz is 294 times faster than software on a dual AMD MP2600+ processor machine at 2.1GHz, and is 2.7 times smaller and 10% slower than the fastest design at 153MHz.

1 Introduction

Reconfigurable devices, such as FPGAs, are now widely used in many applications. The key advantage of this technology is its combination of performance of dedicated hardware with the flexibility of software, without the cost and risk associated with circuit fabrication. Performance can be achieved by exploiting the design's inherent parallelism and by manipulating data at a fine-grain level. As reconfigurable technology makes progress in capacity and performance, there is an increasing need for high-level design methods and tools that can effectively address the growing complexity of hardware design to improve designer productivity. Such tools should enhance design maintainability and portability as system requirements evolve, and should facilitate design exploration so that various trade-offs, such as those in performance and resource usage, can be achieved.

To address these concerns, we develop Haydn [3], a hardware compilation design flow which offers designers a way to capture both cycle-accurate data-paths, and high-level behavioral designs. Both manual and automated design can be used separately or in combination, so that one can achieve the best compromise between development time and design quality; some of our automatically-generated designs are comparable in performance to hand crafted designs.

This paper presents four new developments for Haydn that significantly enhance this approach:

1. a design methodology that supports both cycle-accurate and behavioral levels, in which developers can move from one level to the other (Section 3);
2. an extended scheduling algorithm which supports operation chaining, pipelined resources (with different latencies and initiation intervals), forwarding technique for loop-carried dependencies, and resource sharing and control (Section 4);
3. a new design flow that can be customized with a script language, and can provide extended simulation capabilities for the RC2000 board (Section 5);
4. evaluation of our approach using five case studies: 3D Free-Form Deformation, Gouraud shading, Fibonacci series, Montgomery multiplication, and 1D discrete cosine transform (Section 5).

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

2 Motivation and Related Work

Hardware synthesis tools tend to fall into two distinct camps, namely *cycle-accurate* and *behavioral*-based approaches. Each has its own benefits and drawbacks.

The behavioral approach usually employs a hardware description language that is similar in syntax and semantics to popular software application languages, such as C. The goal of behavioral hardware compilers is to derive one or many hardware implementations from a single high-level description (a process known as *high-level synthesis*), abstracting from low-level details such as timing and resource utilization to allow developers to focus on algorithmic details. One common optimisation technique is pipelining, which makes designs run faster and also reduce power consumption [17]. Since an algorithm can be implemented in a number of ways, behavioral tools can provide an annotation facility for describing constraints and restrict this large design space. Thus the behavioral approach provides several advantages, namely: **(i)** ease of use for software developers, **(ii)** high-productivity for design implementation, and **(iii)** maintainable designs.

However, the behavioral approach has a major drawback: hardware synthesis is performed with little human guidance. High-level synthesis often suffers from lack of user control and transparency over the implementation process. This black-box approach [11] has three unfortunate consequences: **(i)** behavioral constraints can only guide the synthesis process to a limited number of points in the design space and thus it might not be possible to find a suitable design; **(ii)** in the event of generating an unsuitable design, there is little a designer can do, except to play around with behavioral constraints or wander into the intricacies of the generated design; and finally **(iii)** it is difficult to understand the impact of high-level transformations on the resulting design and how they affect different design tradeoffs, such as execution time and resource usage.

Cycle-accurate description languages (such as those based on RTL), on the other hand, give developers more control over low-level implementation details. At this level of abstraction, developers are able to make decisions that would be left to the compiler in a behavioral approach. This allows developers to fine-tune their hardware implementations to achieve an optimal solution. However, cycle-accurate design methodology can have two major disadvantages over high-level synthesis, namely *low productivity and poor maintainability*, which make it highly ineffective for implementing large designs. The lack of productivity is due to the fact that many implementation details and architectural decisions have to be provided at design time. Once these decisions have been made and committed, it is difficult to perform any architectural modifications without developing the whole design from scratch. We believe this to be a serious limitation for three reasons: **(i)** portability is reduced, which means that it is hard to adapt designs to different hardware platforms and constraints; **(ii)** changing design functionality and bug correction is difficult because hardware details are so ingrained with the algorithmic specification, and **(iii)** design exploration is limited and requires a lot of effort. In contrast, behavioral-based designs are highly maintainable as changes can be easily made either through user-defined constraints or in the code itself which is devoid of low-level details.

Goal. Our goal is to develop a hardware compilation approach that combines the advantages of both behavioral and cycle-accurate-based methodologies, so that it can support three important features:

1. Rapid development of *optimized* implementations
2. Design maintainability (portability, adaptability, design exploration)
3. Design quality

So the challenge presented in our research, and which we offer a solution in this paper, is: how to bridge the gap between behavioral and cycle-accurate levels, so that developers can easily roam from one level to the other in order to get benefits from both approaches?

Related Work. There are many examples of behavioral and cycle-accurate approaches in both industry and academia.

Examples of behavioral-based methodologies include: SPC [15], ASC [7], Streams-C [4], Machines [12] and Catapult C [8]. SPC combines vectorisation, loop transformation and retiming to improve design performance. Streams-C, on the other hand, supports highly synchronous communication, but limits pipelining to innermost loop bodies. ASC is also focused on stream computations, and exploits C++ overloading mechanism to automatically generate deep pipelines using the PAM-Blox library. Machines is a programming model that can be implemented in any unmodified object-oriented language. It provides a way for developers to specify coarse-grained parallelism manually, whereas fine-grained details are taken care automatically by the compiler. Catapult C, on the other hand, operates on untimed C/C++, and synthesizes part of the code to RTL. It includes a tool that lets developers generate different RTL implementations based on user-provided constraints, as well as reporting the tradeoff effects in terms of size and speed.

Examples of cycle-accurate description languages are HardwareC [10], Handel-C [2] and RTL VHDL. HardwareC is based on the C-syntax but provides several features relevant to hardware compilation, such as constraint specification, parallel constructs, process and interprocess communication. Handel-C is an extension of ANSI-C, and supports flexible width variables, signals, parallel blocks,

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

bit-manipulation operations, and channel communication. Like VHDL, it gives developers the ability to schedule hardware resources manually, while Handel-C generates the control-path of the design automatically based on its timing semantics.

Our approach is unique in that Haydn combines both cycle-accurate and behavioural design methodologies. This way, developers can opt to use the behavioral approach to rapidly derive a hardware implementation from a high-level design description and constraint annotations. Alternatively, manual intervention can be exerted either at the beginning or at the end of the design cycle to fine-tune a design. We believe that combining both models, manual development and computerized optimizations can be interleaved to achieve the best effect. The next section describes Haydn in more detail.

3 Hardware Design

In the previous section, we identify the benefits and drawbacks of behavioral and cycle-accurate design methodologies. Our approach supports both methodologies. In particular we adopt the following tactics:

Behavioral to Cycle-Accurate. Developers can start the design process by writing a generic C description of the design (see Listing 1) without considering low-level details, such as timing. To automatically derive an optimized design, developers must specify resource and scheduling constraints to guide the source-to-source transformation process. If constraints are satisfied, the source-to-source transformation process generates an optimized design that is ready to be simulated or synthesised to hardware (Fig. 1). This way, developers can rapidly obtain an optimized solution without concerning with low-level details.

Cycle-Accurate to Cycle-Accurate. Once developers get the first implementation, they can improve design performance systematically by modifying constraint parameters (see Listing 2 and Listing 3), running the source-level transformation process and verifying the performance of the generated design. Alternatively, developers can improve performance by manually revising the code without computerized intervention. Hence, both manual and automatic approaches can be used to improve design quality. Furthermore, the ability to automatically transform cycle-accurate designs facilitates design exploration and maintainability. For instance, pipelined designs can be directly accelerated or slowed-down to fit hardware requirements.

Cycle-Accurate to Behavioral. Optimized designs can be very difficult to understand. This is because architectural details are so ingrained with functionality. In this case, we can generate a generic C description that is easier to read, modify, verify and subsequently optimized. This in effect has the advantage of making designs more maintainable to correct bugs and to update its functionality.

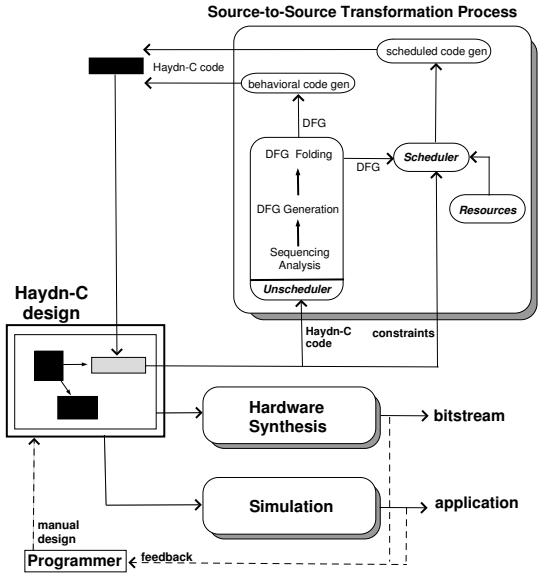

Figure 1: This figure illustrates our hardware compilation approach, which performs source-level transformations, hardware synthesis and simulation of Haydn-C designs. The source-to-source transformation process is guided by annotations in the program that describe design constraints. In particular, this process scans for blocks of code that are enclosed by curly brackets and that are annotated with requests for a particular action, such as scheduling. In this case, the block is removed from the rest of the code, analyzed and the transformed code is put back in place of the original code. Developers can immediately synthesize the new implementation, simulate or perform another transformation, either by manually revising the code or requesting another computerized optimization.

We have developed the Haydn-C language [3] to support this methodology. Haydn-C is based on the Handel-C language, but contains significant differences, which we enumerate next. First, Haydn-C is a component-based language like VHDL. This makes it easy for importing and exporting library blocks (such as IP cores) and working with other HDL tools. Furthermore, our source-level transformation process, which operates under this model, performs a two-way mapping between abstract operators in the program, such as $+$, $*$, and components that specify a particular implementation. Hence, the source-to-source transformation process employs components for generating cycle-accurate (scheduled) designs, and abstract operators for deriving behavioural code. Note that this model is extended to cover user-defined operations that contain a fixed number of input and output ports. Haydn-C also provides a meta-language

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

to support behavioral synthesis, additional data structures such as pipelined FIFOS, and extended macro capabilities (e.g. replicators).

We use Haydn-C to describe hardware designs, and these designs can be transformed, simulated and synthesized to hardware by our compilation design flow (Fig. 1).

Haydn-C does not have any specific language construct to indicate whether a design is behavioral or cycle-accurate. So how do we determine the abstraction level? Before we answer this question, we need to understand the *strict* and *flexible* timing semantics, which we explain in detail next.

3.1 Timing Semantics

Our approach supports two timing semantics. The first, *strict* timing semantics, which is based on the Handel-C language [2], enables us to describe cycle-accurate designs using a few extensions to the C language. These extensions include a `par` statement block to implement parallel structures such as pipelines, and the specification of arbitrary width sizes for variables and expressions. With strict timing semantics, developers can specify the circuit's data-path details and the control-path is inferred automatically. The rules of the model are:

- **i.** All assignment and `delay` statements execute in a single clock cycle.
- **ii.** Expression evaluation and control statements (`if`, `while`) execute within a cycle, contributing to combinatorial delay.
- **iii.** Every statement in a parallel block starts execution simultaneously, and this block only terminates when all threads end.
- **iv.** Each statement in a sequential block starts execution when previous statement terminates, and this block only finishes when last statement concludes execution.
- **v.** There are no data-races. For registers, reads are performed before writes. For wires (signals), reads are performed after writes.

Because hardware synthesis and simulation processes adhere to these rules, developers are able to exert control over the quality of their designs. In other words, users can derive the schedule for a design, and to change the schedule by revising the design.

In order to perform source-to-source transformations, we relax the strict timing rules, which would otherwise be difficult to satisfy. In the *flexible* timing model, every transformation is acceptable as long as it maintains the behavior of the design in relation to its inputs and outputs. This way, for instance, an assignment can be broken down and executed in several cycles to maximize design throughput.

So, the abstraction level is not determined by the design or the language, but by the process which operates on the design. Hence, a design is cycle-accurate when it is synthesized to hardware, since timing is automatically inferred by the strict timing model. On the other hand, the source-to-source transformation process operates at the behavioural abstraction level, since transformations are performed on a data-flow graph, which is devoid of low-level hardware details.

Our source-to-source transformation approach, based on static scheduling, is limited to parts of the program where timing configuration is constant and known at design time (initiation interval and latency). In particular, the scope of our approach is summarized below.

1. Coarse-grained parallelism is specified by users, who can develop their own handshaking protocol or use special communication primitives (such as channel buffers) to synchronize between parallel computational structures.

2. Fine-grained parallelism can be specified with user intervention or extracted automatically by the compiler. Both methods can be used separately or together.

3. Developers are spared from low-level hardware details, and from generating the control-path by hand.

In this paper we focus on fine-grained parallelism. The scope of our approach is not limited to reconfigurable hardware compilation, but can be used for general hardware synthesis. However, we believe that developing maintainable designs, which is one of our stated goals, naturally exploits the benefits of using reconfiguring technology.

3.2 Example

The Haydn-C code in Listing 1–3 is used to illustrate our approach. The purpose is to implement a hardware component that determines the number of solutions for a quadratic equation.

```
1  @resources.set(pipe_mult;UNITS:2; LAT:6; OP:*);
2
3  component quadratic_solutions {
4     in int 32 a;
5     in int 32 b;
6     in int 32 c;
7     out int 2 num_sol;
8
9     code {
10       int 32 delta;
11       { // code to be scheduled
12          @scheduler.run(II: 1);
13          delta = b*b - ((a*c)<<2);
14          if (delta > 0)
15             num_sol = 2;
16          else if (delta == 0) {
17             num_sol = 1;
18          } else {
19             num_sol = 0;
20          }
21       }
22    }
23 }
```

Listing 1: an unoptimized Haydn-C design

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

We begin by writing the interface of the component, which includes the name and port specification (Listing 1, lines 3-7). Next we write the code section (lines 9-22) using C-based constructs. Note that at this point, we can synthesize this code to hardware or optimize it using the source-to-source transformation process. If hardware synthesis is performed then the strict timing rules are enforced, which produce a design with a large combinatorial delay due to the assignment statement in line 13. In particular, this statement contains an expression with two multipliers, and according to the strict timing model, this statement must be executed in a single clock cycle. For this reason we decide to optimize this design automatically by using the source-level transformation process. For the computerized approach, we only need to provide two lines of code, which respectively specify resources to be employed by this process, and the kind of architecture we want to generate. In particular, we supply values for the `pipe_mult` resource attributes in line 1, including the number of units, the latency and specify that it implements the multiplication operator. Next, we identify the block we want to transform (Listing 1, lines 11-21), and annotate the first line of the block (line 12) with an annotation that directs the source-to-source transformation process to generate a design with initiation interval (`II`) of 1, which is shown in Listing 2. This design is fully pipelined, and thus produces a result every cycle.

```
1  @resources.set(pipe_mult;UNITS:2; LAT:6; OP:*);
2
3  component quadratic_solutions {
4     in int 32 a;
5     in int 32 b;
6     in int 32 c;
7     out int 2 num_sol;
8
9     code {
10       int 32 delta;
11       par {
12          @scheduler.run(II: 1);
13          unsigned 32 tmp[3];
14          // ================ [stage 0]
15          pipe_mult[0].in(b,b);
16          pipe_mult[1].in(a,c);
17
18          // ==================[stage 7]
19          tmp[0] = pipe_mult[0].q;
20          tmp[1] = pipe_mult[1].q << 2;
21
22          // ==================[stage 8]
23          tmp[2] = tmp[0] - tmp[1];
24
25          // ==================[stage 9]
26          if (tmp[2] > 0) num_sol = 2;
27          else if (tmp[2] == 0) num_sol = 1;
28          else num_sol = 0;
29          delta = tmp[2];
30       }
31    }
32 }
```

Listing 2: a pipelined design with II=1 generated from Listing 1

Note that only the block from lines 11-21 in Listing 1 is removed, leaving the rest of the code intact, including constraints.

Next, if we wish to use one pipelined multiplier instead of two, we just need to change two lines of code: **(1)** In line 1 of Listing 1 or 2, we set the number of pipelined units to 1 and activate resource sharing by specifying the `RS` argument. **(2)** In line 12, we set the initiation interval (`II`) to 2. Finally, we activate the source-to-source transformation process, which generates Listing 3.

```
1  @resources.set(pipe_mult;UNITS:1; LAT:6; OP:*; RS);
2
3  component quadratic_solutions {
4     in int 32 a;
5     in int 32 b;
6     in int 32 c;
7     out int 2 num_sol;
8
9     code {
10       int 32 delta;
11       par {
12          @scheduler.run(II: 2);
13          { // ================ [stage 0]
14             pipe_mult[0].in(b,b);
15             pipe_mult[0].in(a,c);
16          }
17          { // ==================[stage 3]
18             delay;
19             tmp[0] = pipe_mult[0].q;
20          }
21          { // ==================[stage 4]
22             tmp[1] = pipe_mult[0].q << 2;
23             tmp[2] = tmp[0] - tmp[1];
24          }
25          { // ==================[stage 5]
26             if (tmp[2] > 0) num_sol = 2;
27             else if (tmp[2] == 0) num_sol = 1;
28             else num_sol = 0;
29
30             delta = tmp[2];
31          }
32       }
33
34    }
35 }
```

Listing 3: a pipelined design with II=2 generated from Listing 2

The pipelined design in Listing 3 generates a result every other cycle to accommodate sharing a single multiplier resource to two multiplication operations. Thus this design will be smaller than the previous one, albeit slower. Note that the code from Listing 2 and Listing 3 can be difficult to read. Hence, we develop the abstraction process (Fig. 2), which reverses the effects of scheduling, and generates code similar to Listing 1 (that is, sequential C code), from either Listing 2 or Listing 3.

4 Scheduling Transformations

In this section we focus on unscheduling algorithms (Section 4.1) and scheduling algorithms (Section 4.2) employed by the scheduling process, and how developers can control and guide these transformations (Section 4.3).

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

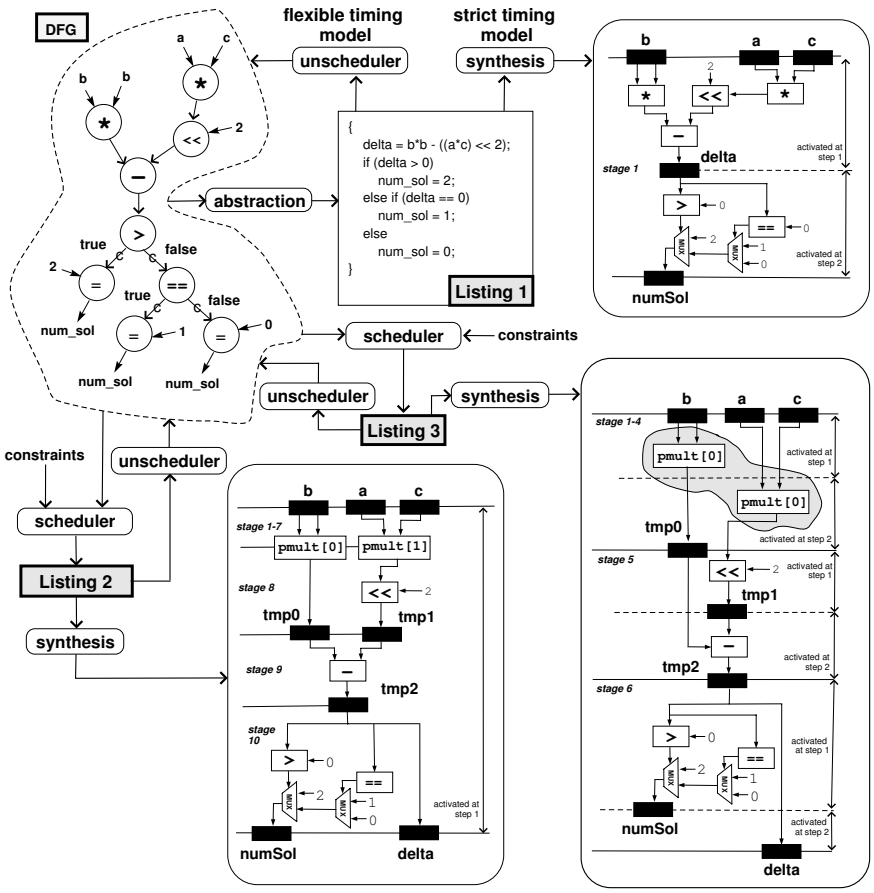

Figure 2: This figure illustrates the Haydn approach. The gray boxes refer to Listing 1, 2 and 3. For each of these designs, we use the **unscheduler** to generate a data-flow graph (DFG), and use **synthesis** to derive a hardware implementation. On the other hand, the **scheduler** and the **abstraction** processes operate on a DFG, and produce a cycle-accurate and a generic sequential C design respectively.

4.1 Unscheduling Algorithms

The unscheduling stage is responsible for generating a data-flow graph (DFG) from a Haydn-C design. A DFG is a source representation where nodes represent program operations, and edges represent the dependencies between them. Our DFG supports four node types and six edge types. Node types include special operations (sink and source), built-in operations (arithmetic, logic and relational), user-defined operations (defined by components) and memory operations (load and store). Edges, on the other hand, can be classified as: true-dependent (read-after-write), anti-dependent (write-after-read), output dependency (write-after-write), input dependency (read-after-read), control dependency and link dependency (between a special node and any other node). Some edges can provide additional information on dependencies between loop iterations. Our implementation supports input dependencies for inferring shift registers from load memory accesses, thus potentially increasing design throughput.

To generate a DFG, we extend existing techniques for collecting data-flow information to support both sequential and parallel Haydn-C designs based on the strict timing model. This procedure consists of three steps. First in *sequencing analysis*, we compute the starting times for each statement enclosed in a given block. The set of statements that execute in the same cycle form a basic timing block. Next, to build a DFG, we compute the information generated and killed in a basic timing block, as well as the information consumed and produced in the next block. The last step, *DFG Folding*, removes all temporary registers and pipelined FIFOs that cause unwanted loop carried dependencies and limit the parallelization effort. At this point, the unscheduling stage is completed and we have derived a DFG from a Haydn-C design. Although the original timing and resource configuration is lost, the functionality is pre-

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

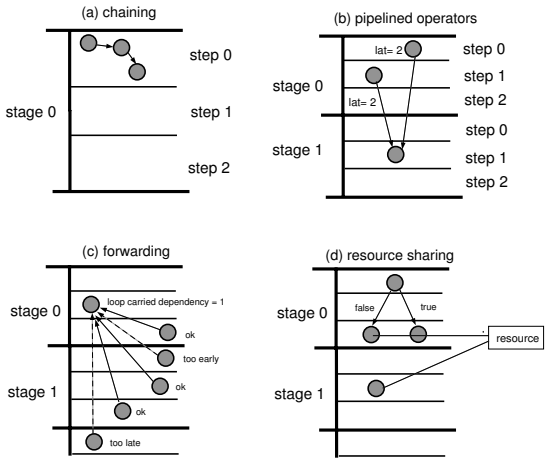

Figure 3: This figure illustrates operation chaining **(a)**, where a sequence of operations are scheduled in the same cycle. The operation in stage 1 **(b)** needs to wait until both sources output valid data. Forwarding **(c)** schedules two nodes with loop carried dependencies, so that its distance matches the position in the pipeline. Two operations can share the same resource **(d)** if they are in different steps, or if they are mutually exclusive and are located in the same stage and step.

served. We can now proceed to the next stage and schedule the DFG. A detailed explanation and description of all unscheduling algorithms can be found elsewhere [3].

4.2 Scheduling Algorithms

The scheduling stage generates a Haydn-C design based on a DFG and given constraints. In our current implementation, we solve the *minimum-latency* resource-constrained scheduling problem, where developers specify available resources and other hardware constraints, such as cycle time, latency and initiation interval. Next, the scheduler assigns a time-step for each DFG node if it is able to satisfy all constraints, including program dependencies. The *minimum-resource* latency-constrained problem will be considered later.

Pipelining. Our scheduling algorithm is based on *list scheduling* [9]. This algorithm can be used to generate a design with user-given initiation interval II_d, which refers to the number of cycles between consecutive inputs and outputs. A pipelined design is composed by a number of stages $(S_0, S_1, ...S_n)$ that run in parallel. For synchronous pipelines, all stages execute in II_d number of steps. The list scheduling algorithm is executed in a number of iterations. At each iteration, we store a list of all DFG nodes whose ancestors have been scheduled and sort the list according to the distance to the DFG sink. For each operation in the list, we find a candidate resource. If there are enough of these units, we ensure that resource binding does not violate any given constraints, such as cycle time and latency. When a candidate resource satisfies all constraints, then it is bound to the operation and is committed to that time-step. We have extended our approach [3] to support **(a)** operation chaining, **(b)** pipelined operations, **(c)** forwarding technique and **(d)** resource sharing, which we describe next.

Chaining. Chaining (Fig. 3**a**) allows two (or more) combinatorial operations in a sequence to be scheduled in the same execution cycle, provided that the overall propagation delay (combined delay sum of all chained operations) does not exceed the cycle time constraint.

Pipelined resources. Our approach supports pipelined resources with different latencies (number of cycles to produce the first result) and initiation intervals, and we assume that that these values are constant and known at compile time. To support pipelined resources, we need to verify the latency and initiation interval constraints. The former ensures that enough cycles have elapsed so that every source (ancestor) operation has valid data available (Fig. 3**b**). The latter makes sure that the pipelined resource with initiation interval II_r can be scheduled in a design with initiation interval II_d. In particular, the following conditions need to be true: $II_r \leq II_d$ and II_d **mod** $II_r = 0$. This condition ensures that consecutive inputs on a specific step always produce a result in the same step throughout the execution of the pipeline. Having resources with different properties helps us achieve different tradeoffs. For instance, pipelined resources with large initiation intervals are often smaller in size. On the other hand, pipelined resources with small initiation intervals are often optimised for speed, but they can exhibit large latencies.

Forwarding. Forwarding (Fig. 3**c**) or bypassing is a hardware technique used to address loop-carried dependencies (feedback cycles) in pipelined designs. In this case, operations in pipeline stages near the end feed data to operations in stages near the front. To ensure that data are generated in time, op_1 (target) and op_2 (source) must be either placed: (a) in the same pipeline stage but in different steps, (b) in step c_1 at stage s_n, and in step c_2 at stage s_m respectively, so that the loop-carried distance is equal to $m - n$, and $c_1 \leq c_2$. Loop carried dependencies can be found in many designs, and this technique can be employed to tackle this problem without slowing down the design or having to deal with out-of-order data [13].

Resource sharing. Resource sharing (Fig. 3**d**) is a powerful technique that binds a single resource to more than one operation. Our approach supports four sharing levels, which include **(a)** no sharing restrictions, **(b)** sharing to a limited number of operations, **(c)** sharing only when in different steps, **(d)** sharing only when in the same step. Note that sharing is only performed when valid, and different re-

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

Table 1: This table describes some of the object-oriented annotations used to describe design constraints and to direct source-level transformations.

Object	Method	Type	Description
resources	set	property	updates resource attributes, such as number of units and sharing level
scheduler	options	property	sets scheduling constraints such as initiation interval and cycle time
	set_smallest	property	configures scheduling and resource constraints to achieve the smallest design (e.g. unrestricted resource sharing)
	set_fastest	property	configures scheduling and resource constraints to achieve the fastest design (e.g. no resource sharing)
	run	action	executes the scheduling process

source sharing levels can be set for each resource. If no sharing restrictions apply, then two operations can share a resource if: **(i)** they are in different steps, or **(ii)** they are in the same step and stage, and are mutually exclusive.

4.3 Controlling Transformations

To control the source-to-source transformation process we use an object-oriented annotation style, which enables users to describe design constraints and to direct source-level transformations. Each annotation is in the form of $@object.method(args)$, where *object* refers to one of the transformation process modules, such as the scheduler or unscheduler. Method refers to either an object property or a particular action to be performed.

These annotations can be placed either in the global scope of the program or in the beginning of each block ($\{\ldots\}$). All annotations located in the global scope are read first, followed by annotations inside each component. If a property-type annotation is found, then the state of source-to-source transformation process is updated in respect of the object specified, and scanning continues to the next annotation. On the other hand, action-type annotations perform code transformations, by removing code from the source and replacing it with the new optimised code. Table 1 shows a list of these source-level annotations and methods. Each method can have an arbitrary number of arguments with syntax: *key0:value0; key1:value1; key2*. Arguments are separated by semi-colons, and each define a key and an optional value.

Developers are responsible to set up a constraint model in which the scheduling process operates. Such model does not reflect a real hardware configuration, as the transformation process does not take into account low level details such as technology mapping, placement and routing. Instead, the constraint model is used to control the optimization process. For instance, a combinatorial multiplier could be set to have a delay of 10, whereas a pipelined multiplier could be set to 5. If developers set the clock period to 5, then in this case no combinatorial multipliers are used. More elaborate schemes can be constructed that can exploit other features such as resource sharing.

5 Implementation and Evaluation

Our design flow (Fig. 4) contains several commercial and public-domain tools. For hardware synthesis, we use our *HyHC* compiler to convert Haydn-C into Handel-C code, which is then processed by DK3 to generate VHDL. Logic synthesis is performed by Sinplify [14] and we use Xilinx [18] tools to generate the bitstream. The host is implemented in C++ and compiled by Microsoft Visual C++.

The simulation process is similar to hardware synthesis, using the same source code. In particular, HyHC is used to generate Handel-C code for simulation that is linked with the host code to produce a single multi-threaded application. We have implemented a simulation library specific for the RC2000 board, which replicates the behaviour of memory banks, local bus, and synchronous primitives for both the host and hardware environments.

Our source-to-source transformation compiler (TC-1) is based on the SUIF framework [16], and we have changed the IR (intermediate representation) to support parallel blocks and arbitrary widths for scalar and array declarations. Also, SUIF IR is extended to support detailed information on the location (such as column, line, size) of each part of the program (blocks, variables, conditional statements, expressions). This is particularly useful for our approach, where code to be transformed is removed and substituted with the optimised code.

We evaluate our hardware compilation approach with five case-studies: 3D free-form deformation (ffd) [6], Gouraud shading (gshade), fibonacci series (fib), Montgomery multiplication (montmult) [1] and 1D discrete cosine transform (dct-1D). We use our source-level tansformation compiler (TC-1) to automatically schedule each design to achieve different tradeoffs in size and speed. For these

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

Table 2: This table presents five case studies to illustrate design trade-offs; all designs target the Xilinx XC2V6000 device. All implementations are generated automatically, however variables in dst-1d_1 are replicated manually to ease routing. Corresponding software functions are developed on a dual-Athlon AMD MP 2600 system at 2.1GHz for comparison.

design	slices	max freq (Mhz)	throughput(Mb/s)	Performance[1]	description
ffd_1	965	164	2.77	6.6x slower, 2.8x larger	Sequential with 5x pipelined LUT multipliers(with lat=5)
ffd_2	404	137	2.32	7.8x slower, 1.1x larger	Sequential with 1x pipelined block multiplier(with lat=3)
ffd_3	1919	158	5.35	3.39x slower, 5.6x larger	Sequential with 10x multipliers
ffd_4	918	153	5973	328x faster, 2.7x larger	fully pipelined (lat=32 cycles) with 5x multipliers
ffd_5	339	137	5352	294x faster, *smallest*	fully pipelined (lat=26 cycles) with 5x block multipliers
ffd_SW	–	–	18.18	–	software version
gshade_1	109	252	4534	24.9x faster, *smallest*	pixel = 8 bits
gshade_2	699	207	8280	45.5x faster, 5.5x larger	pixel = 24 bits
gshade_SW	–	–	182	–	software version
fib_1	118	192	879	2.27x slower, 1.3xlarger	non-pipelined, II=7
fib_2	90	148	4747	2.3x faster, *smallest*	fully-pipelined, lat=2
fib_SW	–	–	2000	–	software version
montmult_1	88	221	2647	3.47x faster, *smallest*	8 bit operands
montmult_2	287	175	8400	11x faster, 3.2x larger	32 bit operands
montmult_SW	–	–	762	–	software version
dct-1D_1	868	173	3322	1.8x faster, *smallest*	pipelined with II=5, lat=12, sharing 5 multipliers
dct-1D_2	2086	166.47	15981	8.9x faster, 2.4x larger	fully pipelined, lat=12
dct-1D_SW	–	–	1778	–	software version

[1] performance relates to: (1) software version for speedup (2) smallest hardware design for resource usage

case studies, we attain different performance gains by:

- using specialized resources, such as block multipliers (*ffd*);
- resource sharing (*ffd, dct-1D*);
- using forwarding technique (*fib*);
- parameterizing bit-widths (*gshade*).

The results are shown in Table 2 and have been obtained using the design flow shown in Fig. 4.

Resource binding can affect overall performance, so mapping the right resources to program operations is essential to get the best performance and tradeoffs. In the *ffd* case study, we use two types of multipliers: LUT (look-up table based) multipliers and block multipliers. Unsurprisingly, using block multipliers helps reduce the overall number of look-up tables used, but routing delays can affect cycle delay. We are able to derive both designs automatically by specifying a different number of units for block and LUT multipliers. For instance, to generate a fully pipelined design with block multipliers, we just need to set the number of units for block multipliers to match the number of multiplier operations in the program (`@resources.set(blockmult;UNITS:5)`) and set the number of units for LUT multipliers to zero (`@resources.set(lutmult;UNITS:0)`).

Sharing can help reduce resource usage. In ffd_2, five multiplication operations share a single pipelined multiplier. To achieve this design, we just need to set the number of LUT or block multipliers units to 1 (see above), and run the scheduler process with initiation interval of 5 (`@scheduler.run(II:5)`). This means that the design requires 5 cycles to generate one result, but resource usage is decreased significantly.

Forwarding is used to implement a fully pipelined design that generates the fibonnacci series ($x[i+2] = x[i+1]+x[i]$) on a single RAM. The design has two loads and one store, and therefore cannot be fully pipelined on a single-port RAM. To overcome this limitation, we employ a dual-port block RAM with read-before-write configuration. Furthermore, these two loads can be combined into one using shift-registers. Forwarding ensures that the *store* operation ($x[i+2]$) is in pipeline stage 2, and the load operation ($x[i]$) is in stage 0, as the distance between both arrays is 2. Note that in this case, the fully pipelined version is faster and consumes less resources than the slowest version, as the non-pipelined version requires more resources to implement the control path.

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 IEEE

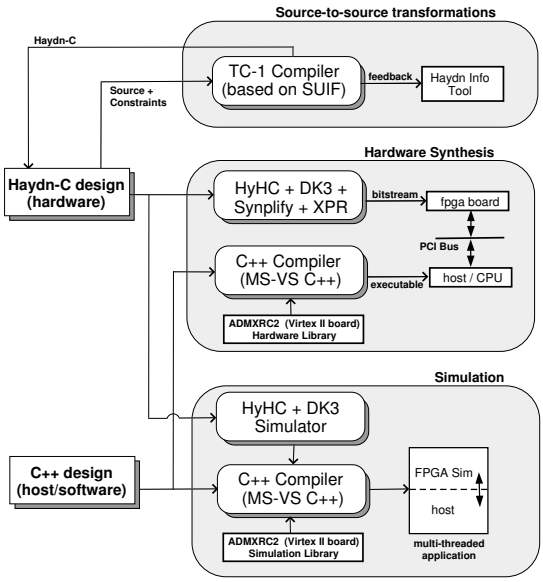

Figure 4: This figure illustrates our design flow, which performs hardware synthesis, simulation and source-to-source transformations. The Haydn-C language is used to develop hardware designs, whereas C++ is used to implement the host which runs on a CPU. For hardware synthesis, a bitstream and the host application are generated, and both communicate with each other using the PCI bus. Simulation involves linking both hardware and host designs into a single multi-threaded application to simulate the behaviour and communication protocols.

6 Conclusion

This paper describes Haydn, a hardware compilation approach that enables designers to combine cycle-accurate descriptions with behaviorial descriptions. Early results show much promise: many efficient designs with different tradeoffs have been developed rapidly and automatically with our approach. Current and future work includes: (a) extending the source-level transformation approach to support simulation to help designers decide what parts of the program would benefit from cycle-accurate description; (b) supporting dynamic scheduling and communication primitives that can abstract from low-level handshaking details; (c) verifying the correctness of our automated transformations.

Acknowledgements

The support of *Fundação para a Ciência e Tecnologia* (Grant number SFRH/BD/3354/2000), UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 31409 and GR/R 55931), Celoxica Limited and Xilinx, Inc. is gratefully acknowledged.

References

[1] V. Bunimov, M. Schimmler and B. Tolg, "A Complexity-Effective Version of Montgomery's Algorithm", in *Workshop on Complexity Effective Designs*, May 2002.

[2] Celoxica Ltd, `http://www.celoxica.com/`

[3] J.G.F. Coutinho and W. Luk, "Source-Directed Transformations for Hardware Compilation", *IEEE Int. Conf. on Field Prog. Tech.*, 2003.

[4] J. Frigo, M.B. Gokhale and D. Lavenier, "Evaluation of the Stream-C C-to-FPGA Compiler: An Applications Perspective", in *Proc. of Int. Symp. on FPGA*, 2001.

[5] M. B. Gokhale and J. M. Stone, "NAPA C: compiling for a hybrid RISC/FPGA architecture", in *Proc. FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1998.

[6] J. Jiang, W. Luk and D. Rueckert, "FPGA-based Computation of Free-Form Deformations", *IEEE International Conference on Field Prog. Tech.*, 2003.

[7] O. Mencer, D. J. Pearce, L.W. Howes and W. Luk, "Design Space Exploration with A Stream Compiler", *IEEE International Conference on Field Prog. Tech.*, 2003.

[8] Mentor Graphics, `http://www.mentor.com/`

[9] G. Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill Edition, 1994.

[10] G. Micheli, D. Ku, F. Mailhot and T. Truong, "The Olympus Synthesis System for Digital Design", *IEEE Design and Test*, pp. 37–53, Oct. 1990.

[11] R. Sharp and A. Mycroft, "A Higher-Level Language for Hardware Synthesis", in *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, LNCS 2144, 2001.

[12] G. Snider, "Attacking the Semantic Gap between Application Programming Languages and Configurable Hardware", in *Ninth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 115–124, ACM Press, 2001.

[13] H. Styles and W. Luk, "Exploiting Program Branch Probabilities in Hardware Compilation", *IEEE Trans. Computers*, 53(11), 2004.

[14] Synplicity Inc., `http://www.synplicity.com/`

[15] M. Weinhardt and W. Luk, "Pipeline Vectorization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Feb. 2001.

[16] R. Wilson et al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", *ACM SIGPLAN Noticies*, 29(12), Dec. 1996.

[17] S. Wilton, S-S. Ang and W. Luk, "The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays", *Field-Prog. Logic and App.*, LNCS 3203, Springer, 2004.

[18] Xilinx Inc., `http://www.xilinx.com`

Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)
0-7695-2445-1/05 $20.00 © 2005 **IEEE**