Source-Directed Transformations for Hardware Compilation

José Gabriel F. Coutinho and Wayne Luk
Department of Computing, Imperial College London
180 Queen’s Gate
London SW7 2BZ, England
{jgfc,wl}@doc.ic.ac.uk

Abstract

This paper presents the Haydn-C language and its parallel programming model. They have been developed to support modular hardware design, to improve designer productivity, and to enhance design quality and maintainability. The principal innovation of Haydn-C is a framework of optional annotations to enable users to describe design constraints, and to direct source-level transformations such as scheduling and resource allocation. We have automated such transformations so that a single high-level design can be used to produce many implementations with different design trade-offs. The effectiveness of this approach has been evaluated using various case studies, including FIR filters, fractal generators, and morphological operators. For instance, the fastest morphological erosion design is 129 times faster and 3.4 times larger than the smallest design.

1. Introduction

The rapid advance in integrated circuit technology in general, and FPGA technology in particular, demands design methods and tools for effective exploitation of such advance to produce more complex designs in shorter time. Many developers have recognised the opportunities of using a high-level, software-oriented language for hardware development: improving designer productivity and design quality, and enhancing maintainability and portability as system requirements evolve.

Powerful optimisations, such as those that involve scheduling and resource allocation, are required to map high-level descriptions into efficient hardware implementations. This paper proposes a style for directing such optimisations based on optional annotations in the source code. In particular, our research covers: (a) a new language called Haydn-C and its parallel programming model; (b) a framework of optional annotations to enable users to describe design constraints, and to direct source-level transformations; (c) development of tools to automate such transformations, so that a single high-level design can be used to produce many implementations with different design trade-offs; (d) evaluation of the effectiveness of this approach, using various case studies including FIR filters, fractal generators, and morphological operators.

Hardware compilation systems usually follow two approaches. The first involves systems using software languages, and requires a scheduler to map high-level descriptions into hardware circuits. Systems adopting this approach include SPC [1], Streams-C [2] and SPARK [3]. They take a behavioral description in ANSI-C as input and generate synthesisable VHDL. SPC combines vectorisation, loop transformations and retiming with an automatic memory allocation technique [4] to improve design performance. Streams-C enables developers to explicitly map computation onto an FPGA board and exploits coarse-grained parallelism, while low-level optimisations such as pipelining are performed automatically by the compiler. SPARK is a high-level synthesis framework that applies a list scheduling algorithm with transformations such as speculative code motion and dynamic transformations for improving synthesis results. Such systems usually employ annotations in the source-code and constraint files to control the optimisation process. While they provide good optimisation capabilities from a high-level description, it may not always be easy to control the optimisation process to produce the most desirable trade-offs.

The second approach involves systems for which the source language describes the design specification at cycle-accurate level. Such systems include the Handel-C language [5] and VHDL. Handel-C is an extension of ANSI-C, and supports flexible width variables, signals, parallel blocks, bit-manipulation operations, and channel communication. Like VHDL, it gives developers the ability to schedule hardware resources manually, although Handel-C will generate the control-path of the design automatically based on its semantics. While these languages enable users to exert fine control over the way that their designs are optimised,
Our approach is intended to combine the advantages of the two approaches above: supporting high-level design and optional fine control of design optimisations to achieve the best effect. The remaining sections are as follows. Section 2 provides an overview of our approach. Section 3 introduces Haydn-C and its programming model. Section 4 describes the source-to-source transformations and the related user-directed framework, while Section 5 covers their automation. Section 6 illustrates our hardware compilation system and its evaluation. Finally, Section 7 presents conclusion and future work.

2. Overview

Our objective is to build a compilation system that supports three important and related aspects of hardware design:

1. Optional manual allocation and scheduling
2. Rapid development of optimised implementations
3. Design maintainability

Our compilation system (Fig. 1) works with designs written in Haydn-C. The Haydn-C language combines elements from VHDL, Handel-C and object-oriented languages, and is geared to achieve the above three aspects of hardware design. We use two models to interpret Haydn-C designs: the strict timing model for simulation and hardware synthesis, and the flexible timing model for source-to-source transformations.

The strict timing model is based on the Handel-C timing model [5], which states, for instance, that an assignment statement to a register executes in a single cycle. Using this model, developers can specify cycle-accurate description of their designs, such as building parallel structures and pipelines using Handel-C’s par block statement, or sequential designs using the semantics of ANSI-C. Hence, the strict model offers manual allocation and scheduling, which gives users the ability to assign hardware resources to program operations, and to arrange these resources in time order. This is, of course, an advantage over high-level sequential languages, which only describe a design at algorithmic level. With manual control, developers are not dependent on the quality of the result produced by automatic optimisations, and they can manually fine-tune their designs before hardware synthesis if desired.

However, performing such manual optimisations can take a long time and is error-prone. Haydn-C designs can be scheduled and rescheduled automatically using the flexible timing model, which combines the unscheduler and the scheduler processes. Unscheduling (Fig. 1a) takes a parallel or sequential Haydn-C design as input and builds a dataflow graph (DFG) representation of the design where nodes represent program and control operations, and edges represent the dependencies between them. Scheduling (Fig. 1b), on the other hand, takes a DFG and generates a Haydn-C design that satisfies user constraints and conforms to the strict timing model. These scheduled designs, also known as concrete designs, can now be simulated (Fig. 1c) and synthesised to hardware (Fig. 1d). If users are not satisfied with the result, they can modify the constraints, which are embedded in the source program, and repeat the transformation process as many times as required. Alternatively, developers can optimise the code themselves. Note that in the flexible timing model, the one-cycle per statement rule does not apply, and the challenge of this model is to preserve the semantics of the strict timing model, so that any transformation performed does not lose the behavior of the original program. Our compilation system also supports loop transformations (Fig. 1e), such as unrolling and tiling, which can be performed automatically to improve design performance as well as adjusting the circuit size to given hardware resources [1].

As we have seen, designs can be adapted to new constraints automatically under the flexible timing model, making these designs highly maintainable. For instance, we can accelerate a pipelined design when more resources are present, or slow down otherwise. The unscheduler can also generate a sequential C design (abstract design), shown in Fig. 1f, which does not take into account user constraints nor the strict timing model. In practice, this allows us to...
extract the behaviour of a pipelined design, making it easier to correct and update the functionality of the algorithm. Hence, our approach also facilitates verification, since the correctness of abstract designs can be established more easily than concrete designs: as long as our transformations are correctness-preserving, users can be confident that the resulting hardware would also be correct.

3. The Haydn-C Language

In this section we present the Haydn-C language and its parallel programming model, which we use for simulating and implementing hardware designs, as well as controlling source-to-source transformations. Developers can port their software code to Haydn-C without much effort, since the semantics and syntax are similar to those of ANSI-C. Furthermore, the language offers several extensions to ANSI-C, such as explicit parallel block statements, arbitrary bit size widths, pipelined FIFO structures and other constructs to exploit the underlying hardware. In particular, Haydn-C includes elements from Handel-C, VHDL and object-oriented languages.

Fig. 3 shows a comparison between Handel-C and Haydn-C features. The main difference between the two languages is that Haydn-C, like VHDL, is component-based. Hence, users can describe their designs as a set of distinct components that are developed independently and then plugged together through its port interface (Fig. 4b). The port interface defines the names and sizes of the input and output ports. The main advantage of the component-based approach over C subroutines (functions) is that there is no distinction between using components written in Haydn-C and external components defined in a component library (Fig. 1d). This black-box approach makes it easy to interchange between internal and external components. To define a component that accesses an ex-
Figure 3. Features found in Handel-C and Haydn-C languages. Handel-C is designed to facilitate porting ANSI-C applications to hardware with features like C-subroutines, structures, and pointers. Haydn-C, on the other hand, combines component-based facilities of VHDL, Handel-C’s parallel programming model (strict timing model), and object-oriented structures for hardware implementation and for controlling the source-to-source transformation processes.

A component in a Haydn-C design, we specify the `EXT=external_component_name` tag in the component attribute section, and ensure that the external component name and port definition match the information defined in the component library (Fig. 4a). The code section of external components is left for simulation.

Haydn-C also supports the `entity` construct to specify component templates. These templates can be optionally parameterised to generate components with different behaviors and/or structures based on a single description. While component definition implicitly instantiates one copy of the component, users are required to instantiate templates to generate one or many components. Fig. 4 shows how entities and components are defined, and how entities can be instantiated.

The previous section introduces the `strict` and `flexible` timing models. The `strict` timing rules enable the user to derive the schedule for a design, and to change the schedule by revising the design. More specifically, they give developers the ability to manage the number of clock cycles for the entire design. Furthermore, it gives users some degree of control over cycle time. Because each assignment executes in a single cycle, developers can reduce cycle time by splitting a statement into two or more statements to reduce the combinatorial delay of the circuit. The rules of the `strict` timing model include [5]:

i. All assignment and delay statements execute in a single clock cycle.

ii. Expression evaluation and control statements (if, while) execute within a cycle, thus contributing to combinatorial delay.

iii. Every statement in a parallel block starts execution simultaneously, and the block terminates when all threads end. Statements in a sequential block start execution when the previous statement terminates, and the block terminates after the last statement finishes execution.

iv. A value written to a register is only available in the next

Figure 4. The definiton of the `pipe_mult` entity is shown in (a). The syntax of an entity definition is similar to a component definition (b), except that a list of optional generic arguments can be specified. To generate components from an entity, we invoke the entity name, and pass down the name of the component, the number of components to instantiate (optional) and the actual generic parameters. In (b), for instance, the generic arguments include the name of the external component, the size and the latency of the pipelined multiplier. A component definition, on the other hand, implicitly instantiates the component. To invoke a component (c), we can either treat it as a subroutine, and specify the input and output arguments (e.g. `myfunc` component); or treat the component as an object, and use the `in` method to input data, and call the method with the output port name to get the result (e.g. `mult8<8>` component). The `design` keyword defines the top-level component.
clock cycle. A value written to a wire is available in the current cycle.

The strict timing model treats arithmetic and logical operators as combinatorial circuitry, since they are used as part of an expression. By default, scalars are treated as registers and arrays as a set of registers. However, variables can be also be associated to wires (signals in Handel-C), on-chip RAMs, and external memory banks.

Although Haydn-C is not an object-oriented language, it adopts an object-oriented style to manipulate components, pipelined FIFOs and the source-to-source transformation process. An instantiated component is treated as an object in the program, and users can use the in method to input data and results can be obtained from the component using the method with the output port name (Fig. 4c). Pipelined FIFOs are structures that carry data across pipeline stages. Data can be pushed on top of the queue using the in method. The DEPTH method returns the size of the FIFO and each stage can be accessed using the array notation. An example of a pipelined FIFO is shown in Fig. 4a, where a FIFO is used to simulate an external pipelined multiplier based on its latency. The timer object can be used to count the number of cycles associated with a particular section of code. Developers can use this object to profile their designs in a runtime environment, both in simulation and in hardware. The source-to-source transformations and the objects used to control them are covered in the next section.

4. Controlling Transformations

The previous section presents the Haydn-C language and the strict timing model, which gives users the ability to manually experiment with different allocation and scheduling strategies, such as to customise pipelined designs.

However, in order to increase both design productivity and maintainability, we devise a transformational approach that changes the timing of a design while preserving its behavioral (Fig. 2). Our transformation-on-demand scheme (Fig. 5a) can generate several types of designs at the request of the user. Transformations include scheduling sequential, non-pipelined and pipelined designs; and loop transformations, such as unrolling and tiling. Developers can therefore explore different types of optimisations to find the best tradeoffs in resource usage and execution time.

The transformation process can be controlled using special predefined objects. These include the scheduler, unscheduler, looptrans and resources objects, which interface the scheduling, unscheduling, loop transformation and resource manager processes respectively (Fig. 5b). For the rest of this section, we focus on the scheduler and resource manager objects to illustrate the source-to-source transformation scheme.

Figure 5. Our compilation system performs a transformation-on-demand scheme, which targets block statements in the source-code (a). Developers can control this process through special objects, in which invoking its methods (functions) triggers an action or an update in the transforming process state (b). To transform a block statement, we invoke one of the action methods, such as the pipeline method of the scheduler object, in the first line of the block statement we want to transform (c). Consequently, the block is substituted with the transformed code, leaving the rest of the source intact, including all constraints defined outside of the block being transformed.
To schedule a design, users need to specify available resources. This can be achieved through the resource object, which defines the main resource scope. Much of the system works on default values, which the user can change to achieve control over the results. In this case, the scheduling is performed using resources available in the main scope if an alternative resource scope is not specified. To insert or update a resource, we use the op method (Fig. 5b). Resources can be declared with several attributes, such as the initiation interval (II), the latency, the number of slices, cycle time and the number of units. The initiation interval (II) is the number of cycles between consecutive inputs. The latency corresponds to the number of cycles until the first result is output. Components defined or instantiated are automatically inserted in the resource list. The new scope method can be used to create a new scope, which can optionally inherit all resources from the main scope.

Another constraint required for scheduling is the maximum clock period, where the default value can be set using the clock method of the scheduler object. Fig. 5c shows how a block statement is pipelined. We invoke the pipeline method of the scheduler object in the first line of the block statement, and the compilation system substitutes the transforming block with the pipelined implementation if it can satisfy all the constraints. These constraints can be declared outside the block being transformed, or passed down as arguments to the pipeline method. The latter include the latency, throughput, maximum clock period (which overrides the default value) and a resource scope (which overrides the main resource scope).

Note that resource attributes and design constraints, such as the number of slices and maximum clock period, are not meant to be accurate or to produce accurate results, since the system does not take into account routing delays and the control path. However, it provides a simple way to control the scheduling and the allocation process. For instance, in Fig. 5c, the scheduler allocates a pipelined multiplier (Fig. 2f), since the combinatorial multiplier has a delay of 25 and would thus violate the maximum clock period constraint, which is set to 20. The pipelined multiplier (mult16), on the other hand, has a delay of 5 as shown in Fig. 4a.

5. Automatic Transformations

In this section we discuss the unscheduling and scheduling phases of our transformational system (Fig. 2).

The aim of unscheduling is to generate a partial-order representation of all program operations. This representation omits timing and scheduling information from the original design while preserving its behavior. Unschedule contains three stages (Fig. 4a): Sequencing analysis, DFG generation and DFG folding. The following describes each of these stages in more detail.

- **Sequencing Analysis.** The first stage of unscheduling computes the starting and ending times of all statements enclosed in the block being transformed, which can include an arbitrary number of parallel and sequential computations. Timing is represented by a set of time-step tags. Each tag, of the form \( n: g_1, g_2, \ldots, g_{2n}, g_{1n}, g_{2n}, \ldots, g_{nn} \), identifies a time step of value \( n \) if every conditional guard \( g_x \) yields true and every conditional guard \( g_y \) is false. The sequencing analysis algorithm is described in detail elsewhere [8].

- **DFG Generation.** In this stage we generate a dataflow graph (DFG) that captures all program dependencies. Our DFG generation algorithm works on the timing information collected in the sequencing analysis stage. Hence, data-flow analysis equations [7] have been extended to compute the information generated and killed in a basic timing block (Fig. 6), as well as the information consumed and produced in the next block. After the DFG is generated, the original timing and scheduling is lost, but we preserve the behavior of the design.

- **DFG Folding.** The final phase of unscheduling removes all temporary registers and pipelined FIFOs from the DFG. This
Figure 7. To perform 1-step folding (a), we select all DFG nodes that originate a loop-carried dependency but are not themselves a target of that type of dependency. Next, we convert these loop-carried dependencies into true dependencies and perform constant folding and forward propagation (b), which removes all selected nodes. We perform 1-step folding until no more nodes can be removed. Note that 1-step folding is only valid if all selected nodes do not change the state of registers declared outside the scope of the block being transformed.

is a required step since these registers exhibit loop-carried dependencies that limit the applicability of the parallelisation process. Our algorithm ensures that registers removed do not compromise the behavior of the design. An example of DFG folding is shown in Fig. 7.

The scheduling process, on the other hand, generates concrete designs from a DFG representation of program operations and dependencies. This process, based on a user-chosen scheduling algorithm, assigns a time-step for each DFG node while satisfying available resources, clock rate, and dependencies between operations (Section 4). The scheduling algorithms used in our system are based on list scheduling [9]. Memory optimisations, such as combining equivalent array accesses into shift registers, are performed automatically to maximise available parallelism [4].

6. Implementation and Evaluation

Our hardware compilation system contains several tools. For simulation, we use the GNU m4 preprocessor [10] to compile Haydn-C designs into Handel-C, which is then passed on to the DK2 [5] simulator. External components can be simulated if the implementation body is provided (Fig. 4a). The m4 preprocessor is also used in hardware synthesis to compile Haydn-C code into Handel-C code. We then use DK2 to produce VHDL, and use Synplify [11] to generate EDIF format files. The final bitstream is produced by Xilinx [12] tools. Designs generated by our system can be used to create component libraries or target RC1000-PP boards [5]. Our source-to-source compiler (TC-1) is built upon the SUIF [13] framework. We have extended the SUIF parser to accept Par blocks and arbitrary widths in scalar and array declarations. This compiler substitutes a selected block with the resulting transformation, giving feedback on the actions performed, and providing an undo command to restore the original design.

We evaluate our hardware compilation approach using six case studies. These include an 8-tap FIR filter, mandelbrot fractal design, 3x3 erosion morphological operator, vector inner product calculator, matrix multiplier, and a 2x2 edge detector. We use our transforming compiler (TC-1) to automatically schedule each test design, initially written as an abstract design, into different concrete designs. For all tests, we define the following constraints. Each set of input and output data is assigned to a unique memory bank, and sufficient resources are made available for parallel scheduling. Hence, performance is only limited by program operations and its dependencies, as well as the scheduling strategy used and hardware resources allocated to implement these operations.

Table 1 presents the smallest and the fastest design for each case study. Sequential designs usually consume the least amount of resources, because these resources can be reused. On the other hand, sharing is limited in parallel designs, and further resources, such as pipelined FIFOs, are required to carry data across different pipeline stages. Most of our results follow this trend. For instance, the fully pipelined implementation of the erosion morphological operator runs 129.70 times faster than the sequential version, given 3.28 times the resources. Furthermore, for a 512×512 bitmap image, the pipelined design runs 12.5 times faster than the optimised software version on a dual Pentium III 900MHz processor. Much of the speedup is due to the fact that block RAM and shift registers are used to fetch the 3x3 kernel, reducing from nine read accesses in the original design to a single memory access. Not all designs can benefit from pipelined implementation. For instance, our non-pipelined fractal design (fract1_np.psm) runs slightly faster than the pipelined design (fract1_pip5.psm), whilst occupying
Table 1. Six case studies to facilitate comparison of the trade-offs between the smallest and the fastest implementation. Design names with the seq suffix are sequential, those with the np suffix are non-pipelined and names that terminate with pip1 suffix are fully pipelined. All designs target the Xilinx Virtex XCV2000E-6 device. Execution time represents the time to process a single result when running at its maximum clock rate. The number of cycles to produce a result corresponds to the initiation interval (II). Latency is the number of cycles until the first result is output, and only applies to pipelined designs.

<table>
<thead>
<tr>
<th>Design</th>
<th>Slices</th>
<th>Exec. time (ns)</th>
<th>II (Lat) (cycles)</th>
<th>Max Freq (MHz)</th>
<th>Tradeoff (Space / Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr_seq_cm</td>
<td>957</td>
<td>830.5</td>
<td>36 (→)</td>
<td>43.4</td>
<td>3.3× smaller /</td>
</tr>
<tr>
<td>fr_pip1_pm</td>
<td>3188</td>
<td>9.6</td>
<td>1 (18)</td>
<td>104.4</td>
<td>86.6× faster</td>
</tr>
<tr>
<td>frcnt_seq_cm</td>
<td>882</td>
<td>388.8</td>
<td>13 (→)</td>
<td>33.4</td>
<td>—</td>
</tr>
<tr>
<td>frcnt_np_pm</td>
<td>930</td>
<td>105.7</td>
<td>12 (→)</td>
<td>113.5</td>
<td>3.6× faster</td>
</tr>
<tr>
<td>erosion_seq</td>
<td>330</td>
<td>1432.4</td>
<td>106 (→)</td>
<td>74.0</td>
<td>3.4× smaller /</td>
</tr>
<tr>
<td>erosion_pip1</td>
<td>1148</td>
<td>11.0</td>
<td>1 (23)</td>
<td>90.5</td>
<td>128.7× faster</td>
</tr>
<tr>
<td>innerprod_seq</td>
<td>924</td>
<td>125.2</td>
<td>6 (→)</td>
<td>47.9</td>
<td>1.8× smaller /</td>
</tr>
<tr>
<td>innerprod_pip1</td>
<td>1659</td>
<td>9.9</td>
<td>1 (10)</td>
<td>101.0</td>
<td>12.6× faster</td>
</tr>
<tr>
<td>edgedet_seq</td>
<td>473</td>
<td>214.3</td>
<td>9 (→)</td>
<td>42.0</td>
<td>2.9× smaller /</td>
</tr>
<tr>
<td>edgedet_pip1</td>
<td>1409</td>
<td>10.2</td>
<td>1 (8)</td>
<td>98.0</td>
<td>21.0× faster</td>
</tr>
<tr>
<td>matmult_seq</td>
<td>445</td>
<td>162.8</td>
<td>7 (→)</td>
<td>45.0</td>
<td>2.7× smaller /</td>
</tr>
<tr>
<td>matmult_pip1</td>
<td>1204</td>
<td>10.5</td>
<td>1 (12)</td>
<td>95.0</td>
<td>15.4× faster</td>
</tr>
</tbody>
</table>

References


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.

7. Conclusion

We have described a compilation approach that supports customisation control, design productivity and maintainability. Productivity is improved by a scheduling process, which automatically generates concrete designs that satisfy given constraints from an abstract design. Maintainability is enhanced by the automated unscheduling process, which facilitates design rescheduling manually or automatically, and by the Haydn-C language, which supports modular design and optional control of source-level transformations. Current and future work includes extending our approach to cover additional customisation methods such as bit-width optimisation, and verifying the correctness of our automated transformations.