The Journal of Supercomputing, 32, 119–137, 2005

Customisable Hardware Compilation

TIM TODMAN tjt97@imperial.ac.uk
JOSÉ GABRIEL DE F. COUTINHO jgfc@imperial.ac.uk
WAYNE LUK w.luk@imperial.ac.uk
Department of Computing, Imperial College London, 80 Queen's Gate, London SW7 2BZ, England

Abstract. Hardware compilers for high-level languages are increasingly recognised to be the key to reducing the productivity gap for advanced circuit development in general, and for reconfigurable designs in particular. This paper explains how customisable frameworks for hardware compilation can enable rapid design exploration, and reusable and extensible hardware optimisation. It describes such a framework, based on a parallel imperative language, which supports multiple levels of design abstraction, transformational development, optimisation by compiler passes, and metalanguage facilities. Our approach has been used in producing designs for applications such as signal and image processing, with different trade-offs in performance and resource usage.

Keywords: hardware compilation, pipelining, high-level design

1. Introduction

As hardware designs become increasingly complex, an effective hardware compiler for high-level descriptions is essential to produce good quality designs on time and within budget. This paper explains how customisable frameworks for hardware compilation can enable rapid design exploration, and reusable and extensible hardware optimisation.

The key elements of our approach include:

1. a framework for describing and transforming designs captured as parallel imperative programs,
2. the automation of design transformations using compiler passes,
3. the use of a metalanguage to facilitate the production of compiler passes and other customisations.

Related work includes Streams-C, SPARK, ASC, SPC, Handel-C, and Haydn-C. Streams-C [10], SPARK [11], ASC [15] and SPC [22] take a behavioural description in a language such as ANSI-C as input, and generate target code in a synthesisable form such as VHDL or EDIF. Streams-C exploits coarse-grained parallelism in stream-based computations, 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 trailblazing. ASC adopts annotations to direct optimisations for speed, latency or area. SPC combines vectorisation, loop transformations and retiming with memory allocation to improve design performance. These systems usually employ annotations in the source 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.

A different approach is adopted by Handel-C [5], an extension of ANSI-C, which supports flexible width variables, signals, parallel blocks, bit-manipulation operations, and channel communication. It gives application developers the ability to schedule hardware resources manually, and Handel-C tools generate the resulting designs automatically. Haydn-C [6] is a framework that automates source-directed transformations in Handel-C.

This paper explains how languages and tools similar to those for Handel-C can be customised in various ways. As far as we are aware, our framework is the first that supports such a variety of customisations, for both application developers and compiler developers. Several applications in signal processing and computer graphics are used to illustrate the effectiveness of our approach.

2. Customise compilation: Overview

Recent reconfigurable technology offers many customisation opportunities. For instance, one can adopt both fixed-point and floating-point number representations [1]. Other useful transformations include those that enhance parallelism, pipelining or serialisation. The design flow for a hardware compiler can contain the following stages [15].

- Source analysis and transformation: tasks include domain-specific analysis, precision analysis, loop transformation, memory management, data structure transformation, and architecture selection.
- Architecture generation: instantiate, parametrise, and compose hardware modules; control generation.
- Module generation: facilities for producing hardware modules to support architecture generation.

In our framework (Figure 1), a compiler is used for the architecture generation stage. The source analysis and transformation stage can be performed manually or by a compiler pass. A compiler generator can be used to customise both the compiler and the compiler pass, given suitable customisation descriptions which can be expressed in a metalanguage. The hardware compiler produces Pebble modules which can then be used in a hardware implementation. A more detailed description of the components of this framework is covered in the following sections.

3. Customisable compilation framework

This section describes Cobble, a customisable framework for hardware compilation. Section 3.1 gives an overview of the framework and the customisations. Sections 3.2 and 3.3 then present respectively the source and target languages, Cobble and Pebble, for our compiler.

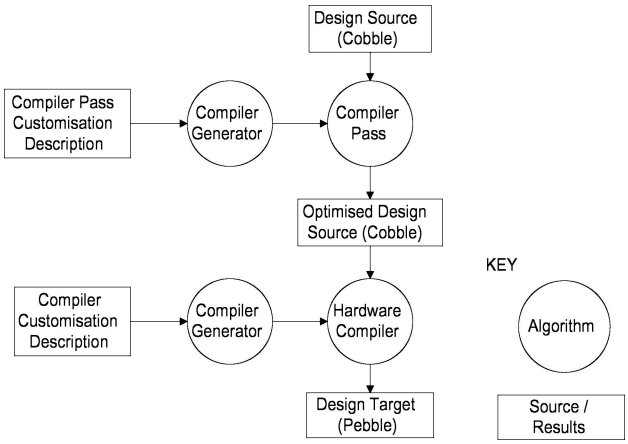

Figure 1. Overview of our framework. The design source are supplied by application developers, while the customisations for the compiler and the associated compiler passes are supplied by compiler developers. The circles contain algorithms and the boxes contain program representation.

3.1. Overview

In our approach, application programs in Cobble [20], a language similar to Handel-C [5], are compiled to Pebble, a language similar to structural VHDL. Unlike Handel-C, Cobble programs can use combinational hardware blocks as expressions. This provides an alternative to Handel-C's bit-level operators which extend C with bit selecting and appending operators; they can be useful for complex bit-twiddling operations, which require the use of recursive macros in Handel-C: Pebble's iterative statements, described in Section 3.3, can provide a more natural description of some operations. Using combinational blocks as expressions can also be a more natural way to use preplaced combinational operators, for which Handel-C adopts a special syntax.

Cobble tool developers can customise Cobble and its tools in three ways, by:

– extending Cobble, the source language,
– customising the compile scheme,
– adding custom transformations at source and target.

As explained in Section 1, various methods can be used in analysing and transforming source programs. The Cobble framework enables compiler developers to experiment with such methods. Cobble defines a metalanguage to simplify specification and development of these customisations, which we describe later in Section 6.

Figure 2 gives an overview of the Cobble framework, showing various opportunities for customisation. The source program, on the left of the figure, may involve the standard Cobble language; it may optionally involve a source language extended by the Cobble framework. Source programs may explicitly call hardware blocks, such as Xilinx cores, or operator implementations—these are shown at the bottom of Figure 2. The source program

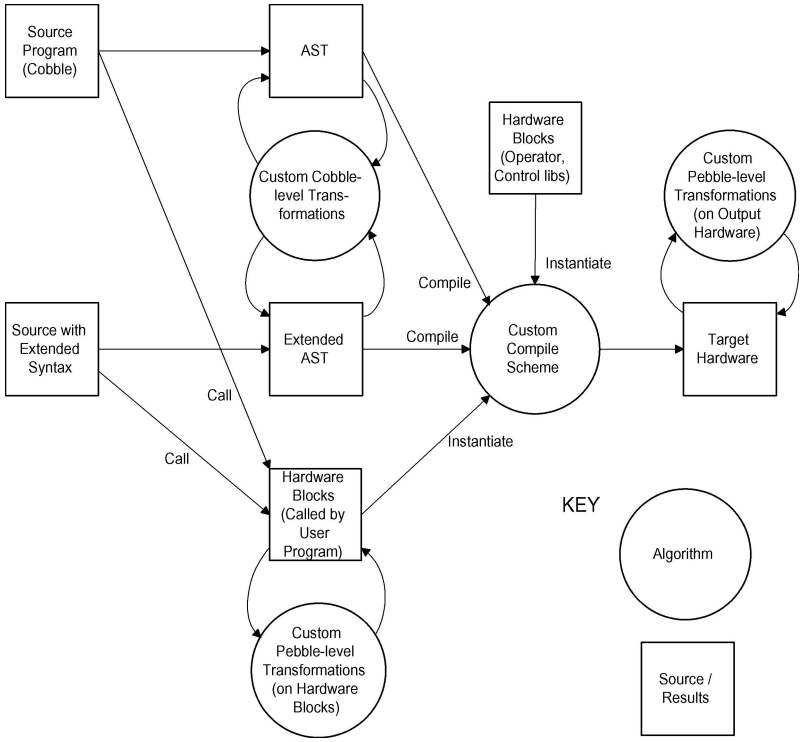

Figure 2. A design tool architecture showing three kinds of customisation: extended input language, custom compile scheme and custom transformations; AST stands for Abstract Syntax Tree. The circles contain algorithms; the boxes contain program representation: source, AST, hardware blocks and compiled hardware.

is initially translated into Cobble's intermediate representation, an AST (Abstract Syntax Tree) with information about the design. Custom transformations can then be applied to the AST representation. At this point, the extended AST can be translated into the core AST, or a custom compile scheme can translate the extended AST into hardware. Note that the instantiated hardware may also undergo custom transformations. A custom compile scheme then compiles the core and extended ASTs into hardware, instantiating hardware blocks called by the user and hardware blocks to implement control flow and operators. Finally, the compiler output hardware can undergo further custom transformations.

3.2. Compilation source: Cobble

This section contains a short introduction to the Cobble language. Cobble is based on a subset of C with extensions for synchronous parallel threads and channels for synchronous communication between them. It has a simple timing semantics: assignment takes one cycle while expressions take none. The timing for all other constructs follows from this—`if` statements take the same number of cycles as the chosen branch, and `while` loops take

```
1  int 4 i;
2  int 8 a[16], b[16], c[16];
3  i = 0;
4  do {
5       par {
6            a[i] = b[i] * c[i];
7            i++;}
8  } while(i);
```

Figure 3. This code shows a basic Cobble loop for multiplying the corresponding elements of two arrays into a third.

the sum of the cycles taken by each execution of their body. Section 5.1 describes these timing semantics in more detail.

The Cobble compiler adopts a simple token-passing scheme [17]. The hardware for each statement is inactive until it receives the token. It then performs its action before passing the token to the hardware for the next statement to be executed. The program is started by an initial token activating the hardware for the first statement of the program.

Cobble allows the programmer to use external hardware blocks such as Xilinx cores [21]. In general, a Cobble program interfaces to these blocks using the same constructs as used for external hardware such as off-chip memories and busses. If the programmer wishes to replace Cobble's built-in multiplication operator with a pipelined preplaced core, they must alter their program to replace an expression with loops to fill, run and drain the pipeline. Cobble also allows combinational external hardware blocks to be used as Cobble expressions.

The example in Figure 3 shows several Cobble features and idioms:

– Line 1 declares `i` with a specific bit-width of 4.
– Line 2 declares three 8-bit arrays. Note that `i` is just wide enough to span the indices of these arrays.
– Lines 4 to 8 show a common Cobble loop idiom. Rather than using a C-style `for`-loop, the loop body (line 6) is run in parallel with the increment of `i` (Line 7).
– Line 8 saves hardware by testing for non-zero `i`. Using `i < 16`, for example, would have required `i` to be one bit wider.
– As a result, the loop takes exactly 16 cycles to run.

3.3. Compilation target: Pebble

Given a Cobble program, the Cobble compiler produces a Pebble program. Pebble [13] can be regarded as a simple variant of Structural VHDL. It provides a means of representing block diagrams hierarchically and parametrically. Pebble has a simple, block-structured syntax. As an example, Figure 5 describes the multiplexor array in Figure 4, provided that the size parameter `n` is 4.

A Pebble program is a block, defined by its name, parameters, interfaces, local definitions, and its body. The block interfaces are given by two lists, usually interpreted as the inputs and outputs. An input or an output can be of type `WIRE`, or it can be a multi-dimensional vector of wires. A wire can carry integer, boolean or other primitive data values. Wires `w1`, `w2`, ... that are connected together are denoted by the expression `connect [w1,w2, ...]`.

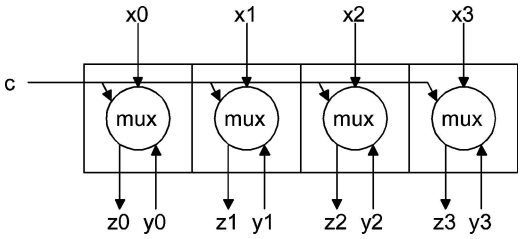

Figure 4. An array of multiplexors described by the Pebble program in Figure 5.

```
BLOCK muxarray (n)
      [c:WIRE, x,y:VECTOR (n-1..0) OF WIRE]
      [z:VECTOR(n-1..0) OF WIRE]
      VAR i;
 BEGIN
  GENERATE FOR i = 0..(n-1)
    BEGIN
      mux [c,x(i),y(i)] [z(i)]
    END
 END
```

Figure 5. A description of an array of multiplexors (Figure 4) in Pebble. The external input c is used to provide a common control input for each multiplexor.

A primitive block has an empty body; a composite block has a body containing the instantiation of composite or primitive blocks in any order. Blocks connected to each other share the same wire in the interface instantiation. For hardware designs, the primitive blocks can be bit-level logic gates and registers, or they can, like an adder, process word-level data such as integers or fixed-point numbers; the set of primitives depends on the availability of the corresponding components in the domain targeted by the Pebble compiler.

The `GENERATE IF` statement enables conditional compilation and recursive definition, while the `GENERATE FOR` statement allows the concise description of regular circuits. To support generic design descriptions, parameters in a Pebble program can include the number of pipeline stages or the pitch between neighbouring interface connections [13]. Different network structures, such as tree- or butterfly-shaped circuits, can be described parametrically by indexing the components and wires.

The semantics of Pebble depends on the behaviour of the primitive blocks and their composition in the target technology. Currently a synchronous circuit model is used in our tools, and special control components for modelling run-time reconfiguration are also supported [13]. However, other models can be used if desired. Indeed Pebble can be used in modelling any block-structured systems, not just electronic circuits.

4. Framework design and implementation

This section describes the design and implementation of the compilation framework introduced in the preceding section. Section 4.1 shows various customisation opportunities in this framework. Section 4.2 sketches an implementation using the Visitor design pattern.

4.1. Customisation opportunities

Compiler developers can customise the Cobble framework in three ways: they can extend the input language, customise the compile scheme, and add custom transformations. Responsibility comes with opportunity–they must check the correctness of the proposed customisations. We show below how the framework enables these customisations.

Extending the source language means that each of the standard compiler phases must be extended to account for the new constructs. We use the ANTLR parser generator [3]; the parser can be extended by writing new alternatives for productions and generating the corresponding AST. The AST itself can be extended by creating a new node to represent the new construct as the sibling or child of the most similar existing node—it is up to the user to choose that node. Alternatively, if the user decides that no new AST element is needed for the new construct, the parser can use existing elements—a source-to-source transformation. The AST is divided into statements and expressions, with in general one AST node per parser rule. We adopt a steeply hierarchical AST, which gives many points to attach a new node for an extended source language. It also allows precise customisation of compile schemes and transformations per kind of AST node.

By a *steep* hierarchy, we mean that each node has few siblings, and many more ancestors than in a shallow hierarchy. The additional ancestors group nodes and their siblings, expressing the commonality between them better than the case when nodes have few ancestors.

Custom compile schemes map standard and extended ASTs to hardware, to support:

- extensions to the input language, unless the extension does not introduce any new AST elements,
- custom concrete protocols, for realising control flow in the compiled hardware,
- custom mappings of existing AST elements to hardware blocks and the connections between them.

Custom compile schemes can be used to implement optimisations. The standard compile scheme maps one AST element to one hardware block. By mapping larger patterns of AST blocks to hardware blocks, we can optimise the mapping of specific patterns. This idea has long been used in software compilers [2].

Consider the following example with two nested loops, the outer one being infinite:

```
// Cobble code
while (1) {
    while (e) {
        s1;
    }
    s2;
}
```

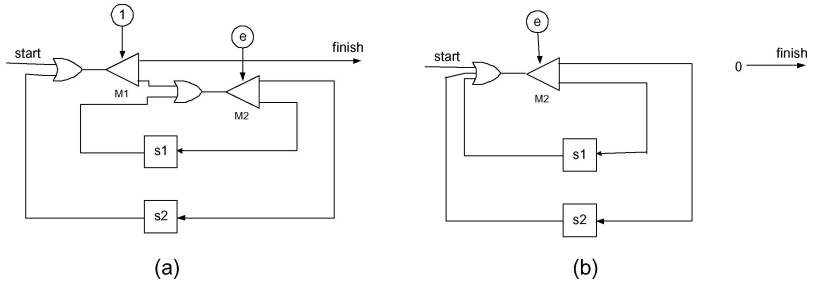

Figure 6. Example of optimisation with a custom compile scheme: (a) with default Cobble scheme; (b) with custom scheme that eliminates redundant demultiplexor M1.

Figure 6(a) shows the output of the standard scheme, with a redundant demultiplexor labelled M1. Figure 6(b) shows the output of a custom scheme to optimise this specific pattern. We could reasonably expect vendor tools to perform this particular optimisation, eliminating useless hardware. They would however be less likely to optimise more complex patterns, which are easier to recognise from the AST than the generated hardware.

Custom transformations modify the program at Cobble and Pebble levels. Cobble-level transformations include conventional compiler transformations such as loop restructuring and strength reduction. Pebble-level transformations can act on Pebble blocks explicitly called by the program or on the compiled hardware output. We have previously shown an example of combined Pebble block and Cobble transformation [21]. Hardware output transformations can implement peephole optimisations.

4.2. Visitor design pattern

We now sketch our implementation that allows Cobble to be customised. Although our current implementation is in C++, we do not depend on any unique features of that language; the same ideas could be implemented in other languages such as Java. We show how our internal AST representation, our internal hardware representation and our use of the Visitor design pattern [9] enable extended input languages, custom compile schemes and transformations.

As we mentioned previously, our AST is steep, which allows it to be extended in many places. This aids input language extensions, as the user can pick precisely where to place the nodes for their language extension.

ASTs used by other frameworks are not as steep. For example, the SUIF 2 [19] hierarchy does not distinguish between logical and arithmetic operators, whereas we do, and we further distinguish logical operators with short-circuit (lazy) and normal (greedy) semantics.

Our AST is implemented as a C++ class hierarchy. The hierarchy follows several C++ design rules to ensure robustness. These rules make it single-rooted, with all non-leaf classes being abstract [16]. We have parallel, related hierarchies for representing types and symbols (symbol table entries), which follow the same design rules.

In conjunction with the Visitor design pattern, the extra levels allow finer distinctions between object types without resolving the actual object type. The idea is to avoid users having to downcast to the actual type wherever possible, simplifying code and providing opportunities for user extensions to the representation without rewriting existing passes.

The Visitor design pattern ([9], page 331) allows new operations to be added to an existing class hierarchy without changing it. The key benefit to our framework of using the Visitor design pattern is in its separation of operations on the AST (custom compile schemes and custom transformations) from classes defining them.

5. Compiler pass: Automating transforms

This section describes extending the Cobble framework with a compiler pass to support transformations that can improve designer productivity and design maintainability.

5.1. Motivation

The Cobble language, described in Section 3.2, provides a cycle-accurate description of a design with its simple timing semantics. It enables application developers to implement an algorithm with various allocation and scheduling configurations.

The allocation problem concerns assignment of program operations to hardware resources; the scheduling problem involves arranging resources in time order. The tradeoff between resource usage and execution time can be obtained through sharing resources and maximising available parallelism, both of which can be expressed manually through the Cobble language.

Developers can manually optimise their designs through Cobble's compilation scheme, which performs a syntax-directed translation from the source program to hardware [17]. This scheme has two important implications. First, application developers can specify which hardware resources are allocated in their designs and how these resources are placed in time order. Second, application developers can assert the quality of their implementation at design time. Even software C programs can be mapped to hardware using this scheme, since resource-binding and timing information not provided can be inferred by Cobble's programming model and semantics. For instance, arithmetic and logical operators are bound automatically to combinational hardware blocks, and scalars and arrays are mapped to registers and groups of registers respectively. Furthermore, every assignment statement runs exactly in one clock cycle, and statements enclosed in a sequential block are only executed after the previous statement has terminated. Manual optimisations often involve the `par{}` construct to specify parallel computations and minimise design execution time. In addition, cycle time can be reduced by incorporating pipelined operators instead of the default combinational implementation (Figure 7(a) and (d)). Moreover, resources can be shared to minimise resource usage, sometimes at the expense of reducing parallelism.

Like Handel-C, the Cobble language and its timing semantics present an advantage over languages that describe designs exclusively at algorithmic level. Application developers using such languages have reduced control over scheduling and allocation-related optimisations, as the compilation process and tradeoff exploration are guided by compilation

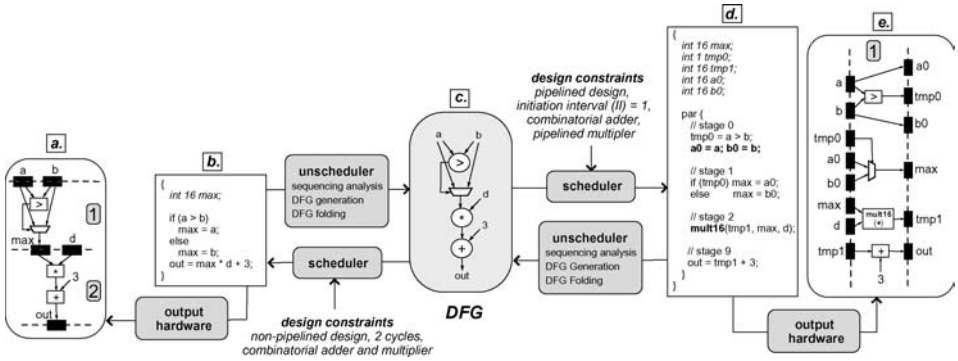

Figure 7. This figure illustrates how Cobble programs are automatically optimised to meet new design constraints. (b) and (d) shows two Cobble designs that are synthesised to hardware ((a) and (e) respectively) using a directed-syntax compilation scheme [17]. Both designs produce the same results given the same inputs, however they have different scheduling and allocation configurations. This affects execution time and resource usage: the hardware design in (a) is likely to consume less resources than the design shown in (e), however the latter should run faster. We combine *unscheduling* with the *scheduling* stage to automatically change the configuration of a design in order to find the best tradeoffs between resource usage and execution time. The *unscheduling* stage comprises three steps: sequencing analysis, DFG generation and DFG folding, and generates a data-flow graph (DFG) from a Cobble program. A DFG represents abstract operations that are partially ordered by dependencies, such as read-after-write and write-after-write. The *scheduling* stage has the freedom to map these abstract operations into existing hardware resources and to place these resources in a new time-order, as long as it does not violate design constraints and dependencies. For instance, the scheduler maps an abstract multiplier operator shown in (c) into a combinatorial multiplier (a). However, if the delay of this combinational multiplier is greater than the cycle time constraint, then the scheduler can use a pipelined multiplier with a shorter delay if available (e).

parameters and source-level annotations. In this case, application developers have to be content with the number and the quality of implementations that the compiler can generate; however the tools may not always produce the desired effect.

Cobble designs can be transformed to meet performance or size requirements. However, performing such transformations entirely by hand has its disadvantages: it can be tedious and error-prone, thus affecting design productivity. Furthermore, once a design has been developed and committed to a particular implementation, it is usually difficult to adapt to new constraints and to change its functionality. Consequently, maintainability is generally low and application developers may have to start the design cycle from scratch.

This paper proposes an approach that overcomes these drawbacks. The proposed approach is intended to combine the advantages of both manual and automated optimisations, to achieve the best implementation in resource usage or execution time, while enhancing both design productivity and maintainability. In the rest of Section 5, we discuss this automated approach and outline its implementation.

5.2. *Unscheduling and scheduling*

We have automated many optimisations, such as minimising resource usage or maximising execution time. Application developers provide the design constraints, such as cycle

time and available resources. Our automated approach includes two stages: ***unscheduling*** and ***scheduling***. The *unscheduling* stage takes a Cobble design and generates a data-flow graph (DFG). The DFG includes nodes that represent program operations, and edges that represent dependencies between these operations. A Cobble program defines a total-order on operations in relation to time, and each operation is mapped to a hardware resource. A DFG, on the other hand, defines a partial-order on abstract operations in relation to its dependencies. The aim of the *scheduling* stage is to define a time-order between DFG operations (scheduling) and map each node to a hardware resource (allocation), without violating dependencies and design constraints.

This method supports automatic rescheduling of a hardware design to different constraints by combining the *unscheduling* and *scheduling* stages. This increases design maintainability, as for instance, a pipeline design can be speeded up if more resources are available, or slowed down if the design is ported to a smaller reconfigurable device. *Unscheduling* contains the following three steps.

- *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 $\boxed{\text{n:}g_{x1}, g_{x2}, \ldots, g_{xn}, !g_{y1}, !g_{y2}, \ldots, !g_{yn}}$, 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 [7].
- *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 [2] have been extended to compute the information generated and killed in a basic timing block (Figure 8), 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 unnecessary operations from the DFG. This ensures that this DFG can be scheduled and rescheduled without compromising its size, behavior and the parallelisation effort [6]. An example of DFG folding is shown in Figure 9.

After the unscheduling stage is complete, we proceed to schedule the DFG. The scheduling algorithms used in our system are based on list scheduling and solve the *minimum-resource* latency-constrained and the *minimum-latency* resource-constrained problems [8]. Furthermore, application developers can control how resources are shared and scheduled in different pipelined and non-pipelined configurations (see Section 7). Memory optimisations, such as combining equivalent array accesses into shift registers, are performed automatically to maximise available parallelism [22].

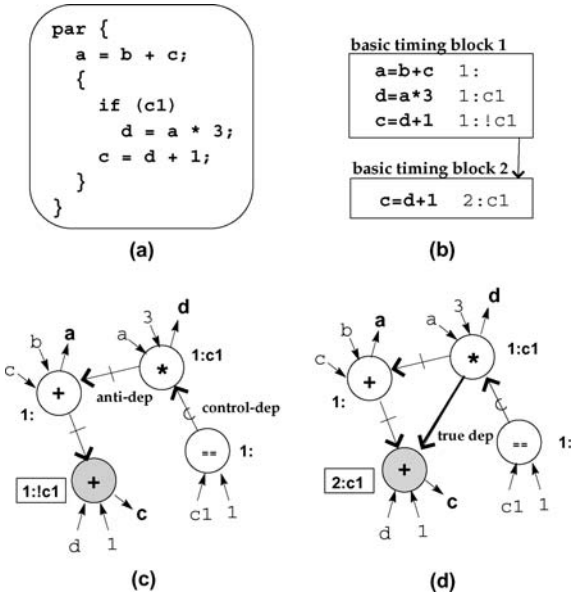

Figure 8. This diagram illustrates the process of generating a dataflow graph (DFG) from a Cobble design. (a) The original Cobble program. (b) A list of basic timing blocks generated from the Cobble program in (a) using the timing information collected by the sequencing analysis stage. (c) The DFG generated after processing basic timing block 1. (d) The final DFG after processing basic timing block 2. To generate a DFG from a Cobble design, we process in sequence each basic timing block, which can contain assignment, conditional and component call statements. Each statement contains a *time-step tag* that indicates when a statement is executed. For instance, `d=a*3` has the time-step tag 1:c1, which means that this statement is activated when c1 is true in time-order 1. For each statement we generate a DFG node, and dependence relations between these nodes are found according to their associated time-step tags. For instance, the shaded adder node and the multiplier, shown in (c), are mutually exclusive in basic timing block 1, because they respectively contain time-step tags 1:!c1 and 1:c1. However, both nodes exhibit a write-before-read relation in basic timing block 2 (d), and thus we include a true-dependence edge.

5.3. Implementation

We develop a compilation pass that performs automatic transformations at source-level for incorporation in the Cobble framework (Figure 10). The goal of this compilation pass is to transform an Abstract Syntax Tree (AST) to meet the required design constraints.

The transforming process starts with the *unscheduling* stage which, as explained before, generates a data-flow graph (DFG) representation of the input design. The first step of *unscheduling* is sequencing analysis. To implement this algorithm, we use the Cobble framework's visitor pattern facility to traverse the AST nodes using depth-first search. When a node is visited, we calculate its starting and ending time and pass this information to its ancestors. The sequencing analysis step terminates when all nodes have been visited. To generate a DFG, we process each AST node in relation to its time-order (calculated in

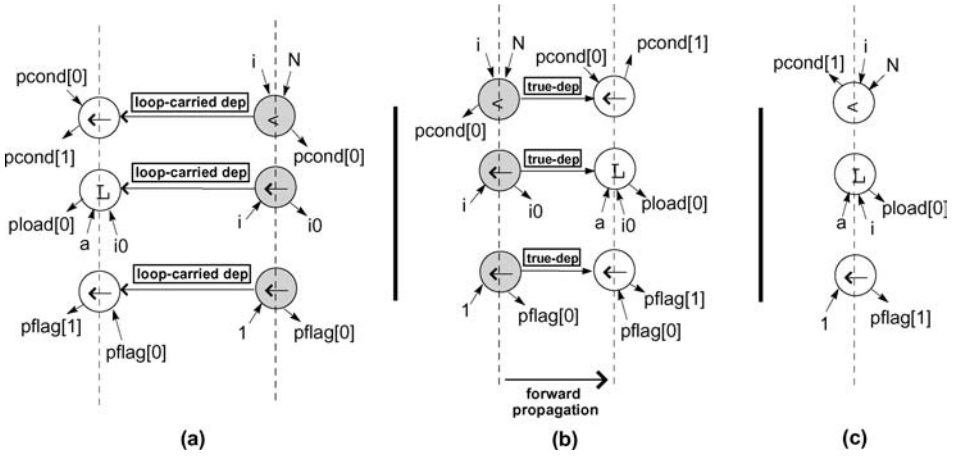

Figure 9. This figure illustrates the *dataflow graph (DFG) folding* transformation, which discards unnecessary operations from the DFG. (a) The original DFG. (b) The DFG after converting loop-carried dependencies into true-dependencies. (c) The DFG after performing forward propagation transformation. Candidate operations for removal include register transfer operations, which are once used to carry data from one pipeline stage to the other. This is an important step, since successive rescheduling of a DFG would otherwise result in an explosion of nodes and edges, limiting the applicability of the parallelisation process. The DFG folding process begins by selecting all operations that can be removed at one time (a) without changing the behavior of a design. The shaded operations exhibit loop-carried dependencies, which we convert to regular true-dependencies, as shown in (b). Finally, candidate nodes are eliminated through forward-propagation (c). Similar transformations have been developed to deal with control edges and propagation of control values across the pipeline.

the previous step), and keep a profile of every symbol used in the program to obtain the correct dependencies between program operations.

The final stage of this compilation pass is *scheduling*, which involves the DFG generated in the previous stage and the design constraints from the input AST. If the scheduler can meet these constraints, then feedback is given about the tradeoffs between the old and the new designs, and a new AST is produced (Figure 10(c)). This new AST can be converted to source-level (Figure 10(d)) or translated directly to hardware (Figure 10(e)). If only part of the design is selected for optimisation (such as a loop), then the new AST is merged with the old AST, substituting only the relevant part of the tree.

6. Metalanguage: Customising tools

This section illustrates how a simple metalanguage, *CML* (Cobble MetaLanguage), can support transformations of Cobble programs. Suitably extended, CML can also be used to produce Cobble compiler passes described in the preceding section.

While the Visitor design pattern provides a way for modular development of design tools for Cobble, in practice it can be complicated to write code using the Visitor design pattern to match syntax patterns—we shall show an example shortly. We have therefore designed a metalanguage to allow users to match syntax patterns without the complications imposed

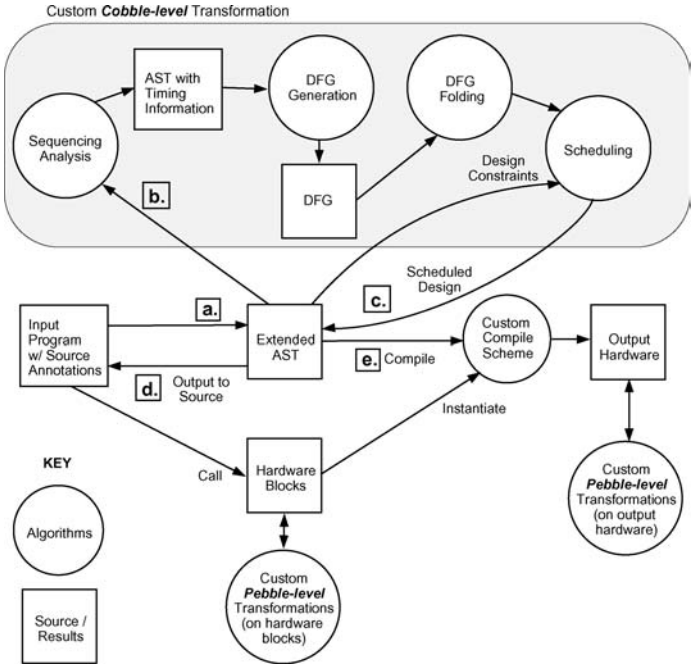

Figure 10. We extend the Cobble framework, shown in Figure 2, to support automatic transformations at source level. The source program is annotated with design constraints, which include available resources and their attributes, and also the allocation and scheduling configuration to be implemented. The source is then translated to an extended AST (Abstract Syntax Tree), which contains all constraints from the original program (a). Next, the transformation pass takes this AST (b) and generates a new AST that meets the design constraints and required configuration (c). This new AST can be converted back to Cobble (d) if the user requires the transformed design at source-level, or it can be compiled directly to hardware using a custom compilation scheme (e). Note that the transforming compilation pass is optional, as users can compile their designs directly to hardware, going from the extended AST (a) to the custom compilation scheme (e).

```
1  class FoldPlus : public ASTVisitor {
2     void visit(Plus *a) {
3        a->getLhs()->accept(this);
4        a->getRhs()->accept(this);
5        if (IntLiteral * i =
6           dynamic_cast<IntLiteral *>(a->getLhs())) {
7              if (i->getValue() == 0) {
8                 a->getParent()->replace(a, a->getRhs());
9              }
10       }
11    }
12 }
```

by the Visitor design pattern. The metalanguage is based on research [4] on descriptions for specifying: (a) syntax patterns to match in ANSI C, and (b) transformations to apply to the matched code, using the SUIF compiler framework [19]. We adapt and extend this work to enable similar specifications of the three kinds of customisation supported by the Cobble framework. Throughout the rest of this section, we refer to the metalanguage in [4] as *CML-pre*.

As a motivating example, consider a compiler pass that folds additions of zero to an expression `x`: `0 + x => x`. An implementation using the Visitor design pattern looks like:

In the above code, we replace a "+" by its right hand side operand if its left hand side operand equals zero. The code works as follows:

- Line 1 declares a class using the Visitor design pattern to visit Cobble's AST.
- Line 2 starts a method to visit `Plus` AST nodes, which are the only kind we need to visit to recognise the pattern `0 + x`.
- Lines 3 and 4 recurse to the operands of the `Plus` node, applying the transformation to them.
- Lines 5 and 6 use a common C++ idiom to try to cast the left hand side of the `+` operator to `IntLiteral`, the AST type representing literal integer values. If this test succeeds, then the left hand side must be a constant integer.
- On line 7, we know that the left hand side is a constant integer and find its value using the `getValue` method of the `IntLiteral` class.
- On line 8, the tests on lines 5, 6 and 7 have succeeded and we have matched the pattern `0 + x`. We instruct the parent of the `Plus` node to replace it with the right hand operand.

CML simplifies the implementation of customisations because it removes the bookkeeping details needed to navigate through the source to transform it. From CML-pre, CML takes the idea of using syntax patterns to specify the source and the corresponding target needed for a compilation or transformation. These syntax patterns consist of source program elements plus *metaelements*, which can match and label any syntax pattern corresponding to a particular type of AST node; for example, the metaelement `expr(1)` would match any expression and give it the label `1`. The labels allow the syntax that matches the metaelement to be copied to the target pattern for transformations, or to be mapped to specific targets for extended source languages and custom compile schemes.

```
1 custom_transform {
2     pattern {
3         0 + expr(1)
4     }
5     generate {
6         expr(1)
7     }
8 }
```

CML-pre maps C language patterns to other C language patterns, corresponding to custom transformations at the Cobble level, which in CML map Cobble language patterns to other Cobble language patterns. CML extends CML-pre to support Pebble-level syntax patterns for custom Pebble-level transformations and custom compile schemes. It also allows new kinds of statement to be added to extend the source language. Using CML, a shorter and clearer description of the above optimisation is shown below:

We explain the above CML code as follows:

– Line 1 starts a `custom_transform` block. Metalanguage descriptions consist of a single block, which must be one of `custom_compile_scheme`, `custom_transform` or `custom_hw_transform`, which respectively correspond to custom compile schemes and custom transformation at software and hardware levels. Each block contains a list of pairs of `pattern` and `generate` blocks, respectively specifying the input and output patterns.
– Lines 2–4 specify the input syntax pattern. Syntax patterns are specified using plain Cobble code plus metaelements; here the metaelement `expr(1)` matches any expression appearing on the right-hand side of the + operator and gives it the label `1`.
– Similarly, lines 5–7 specify the output syntax pattern which will be substituted for any part of the AST matching the input pattern. Here, the expression labelled `1` in the input pattern is copied to become the output pattern, completing the transformation.

The example shows how the CML description removes the book-keeping details needed in the C++ version of the same transformation. It is clear that the CML description is much simpler and more readable than the description involving the Visitor design pattern.

To match more complicated patterns, C++ descriptions using the Visitor design pattern must adopt flags and stacks to record the context of syntax tree elements. This state needs to be maintained for visiting each type in the syntax tree, which can be error-prone. More complicated patterns may also require the C++ code to match them to be spread across several `visit` methods, to several kinds of AST node, further complicating the code needed to match them. CML descriptions hide these complications, and can be systematically translated to Visitor design pattern code, automatically declaring and using this state [20].

CML can be extended to specify the steps needed by the automated transforms in Section 5.

– To specify sequencing analysis, CML would need to be able to read and write user-defined annotations; these annotations would store the time-step tags as lists of conditional guards which can be expressed using the existing AST. A CML pass could then perform the sequencing analysis algorithm.
– To specify DFG generation, CML would need to be extended with (a) iteration constructs, to explicitly iterate over the AST, building the DFG by reading the annotations written by the sequencing analysis, and (b) constructs for building graphs.
– To specify DFG folding, CML would need to be able to specify input and output patterns in graphs, analogously to the existing syntax patterns, complete with metaelements.

– Finally, to specify scheduling algorithms, CML would additionally need primitives to build and iterate over data structures used by those algorithms: for example, list-directed scheduling would require support for list operations in CML.

7. Evaluation

We evaluate our hardware compilation framework using six case studies. These include an 8-tap IIR filter, Julia fractal design, 3 by 3 dilation morphological operator, comb sort algorithm, skeletonization based on Hilditch's algorithm, and the RC5 encoder. We use our automated transformation approach described in Section 5 to automatically schedule each test design, initially written as a straightforward C implementation, into different 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. Hence, performance is only limited by program dependencies, as well as the scheduling strategy used and hardware resources allocated to implement these operations.

Figure 11 presents the smallest and the fastest design for each case study. Design names with the *npN* suffix are non-pipelined with a latency of *N*, and designs with names that terminate with *pipN* suffix are pipelined with an initiation interval of *N*. 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 initiation interval (II) is the number of cycles to produce a result, and also corresponds to the latency of a non-pipelined design. For pipelined designs, latency refers to the number of cycles to produce the first result.

To find the smallest design, we set the scheduler to solve the *minimum-resource* latency-constrained problem and generate a non-pipelined design. Resource sharing, in this case, is set to maximum.

Design	Slices	Exec. time *(ns)*	II (Lat) *(cycles)*	Max Freq *(MHz)*	Tradeoff *(Space / Time)*
iir8_np41_cm	1700	881.7	41 (—)	46.5	**3.7× smaller /**
iir8_pip1_pm	6212	9.8	1 (18)	102.2	**90.1× faster**
jlfract_np15_cm	902	477.7	15 (—)	31.4	—
jlfract_np14_pm	943	135.3	14 (—)	103.5	**3.5× faster**
dilation_np102	325	1416.7	102 (—)	72.0	**3.4× smaller /**
dilation_pip1	1120	11.7	1 (24)	85.2	**120.7× faster**
combsort_np10	610	196.5	10 (—)	50.9	**2.1× smaller /**
combsort_pip4	1310	37.0	4 (8)	108.2	**5.3× faster**
skel_np34	1210	693.9	34 (—)	49.0	**3.0× smaller /**
skel_pip1	3622	9.5	1 (23)	105.0	**72.9× faster**
rc5_enc_np7	409	133.1	7 (—)	52.6	**3.2× smaller /**
rc5_enc_pip1	1310	9.7	1 (4)	103.4	**13.8× faster**

Figure 11. Six case studies to facilitate comparison of the tradeoffs between the smallest and the fastest implementation. Design names with the *seq* suffix are sequential, those with the *np* suffix are non-pipelined and designs with names that terminate with *pipN* suffix are pipelined with throughtput of *1/N*. 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.

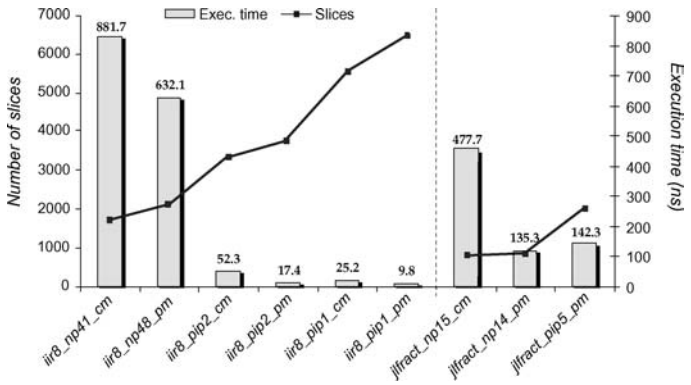

Figure 12. This graph shows how different scheduling and allocation strategies affect the performance and area of two applications. Designs with the *cm* suffix contain combinational multipliers, while those with *pm* contain pipelined multipliers. Results show that designs can achieve higher frequencies by increasing pipeline throughput and using pipelined multipliers, at the expense of increased latency and resources.

The fastest design, on the other hand, is obtained by iteratively solving the *minimum-latency* resource-constrained problem and generating a pipeline design with the minimum initiation interval and no resource sharing. Non-pipelined designs usually consume fewer resources, because there are more opportunities for sharing them. Pipelined designs, on the other hand, are larger in size, since sharing is restricted 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 dilation morphological operator runs 121 times faster than the sequential version, given 3.4 times the resources. Furthermore, for a 512 by 512 bitmap image, the pipelined design runs 12 times faster than the optimised software version on a dual Pentium III 900 MHz processor. Much of the speedup is due to the fact that block RAM and shift registers are used to fetch the 3 by 3 kernel, reducing from nine read accesses in the original design to a single memory access.

Not all designs can benefit from a pipelined implementation though. For instance, our non-pipelined fractal design (*jlfract_np14_pm*) runs slightly faster than the pipelined design (*jlfract_pip5_pm*), while taking up less than half of the resources (Figure 12).

8. Conclusion

This paper describes a customisable framework for hardware compilation that can benefit both application developers and compiler developers. We have identified a multitude of opportunities for customising the source, the compile scheme and the design transformations, for automating source-level transformations by compiler passes, and for facilitating the production of such compiler passes and other customisations based on a metalanguage. Current and future work includes incorporating domain-specific customisations [1, 12] into our framework, verifying the correctness of customisations [14], extending our work to

cover other compilation schemes such as ASC [15], and investigating support for run-time design reconfiguration [18].

Acknowledgments

We thank Ray Cheung and David Thomas for their comments and suggestions. The support of *Fundação para a Ciência e Tecnologia* (Grant number SFRH/BD/3354/2000), UK DTI (project Speedpaint) and EPSRC (Grant number GR/N 66599, GR/R 31409 and GR/R 55931), Celoxica and Xilinx is gratefully acknowledged.

References

1. A. Abdul Gaffar et al. Unifying bit-width optimisation for fixed-point and floating-point designs. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2004.
2. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Series in Computer Science, Addison-Wesley, 1986.
3. ANTLR, `www.antlr.org`.
4. M. Boekhold et al. A programmable ANSI C transformation engine. In *Proc. Int. Conf. on Compiler Construction*, LNCS 1575, Springer, 1999.
5. Celoxica, `www.celoxica.com`.
6. J. G. F. Coutinho and W. Luk. Source-directed transformations for hardware compilation. In *Proc. Int. Conf. on Field-Programmable Technology*, IEEE, 2003.
7. J. G. F. Coutinho, W. Luk, and M. Weinhardt. Optimizing parallel programs for hardware implementation. In *Proc. SPIE ITCom*, 2002.
8. G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. M. B. Gokhale et al. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2000.
11. S. Gupta et al. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. Int. Conf. on VLSI Design*, Jan. 2003.
12. T. K. Lee et al. Compiling policy descriptions into reconfigurable firewall processors. In *Proc. Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2003.
13. W. Luk and S. W. McKeever. Pebble: A language for parametrised and reconfigurable hardware design. *Field-Programmable Logic and Applications*, LNCS 1482, Springer, 1998.
14. S. W. McKeever and W. Luk. Towards provably-correct hardware compilation tools based on pass separation techniques. *Correct Hardware Design and Verification Methods*, LNCS 2144, Springer, 2001.
15. O. Mencer et al. Design space exploration with A Stream Compiler. In *Proc. Int. Conf. on Field Programmable Technology*, IEEE, 2003.
16. S. Meyers. *Effective C++*, 2nd ed. Addison-Wesley, 1998.
17. I. Page and W. Luk. Compiling occam into FPGAs. *FPGAs*, Abingdon EE&CS Books, 1991.
18. N. Shirazi, W. Luk, and P. Y. K. Cheung. Framework and tools for run-time reconfigurable designs. In *IEE Proc.-Computing and Digital Techniques*, May, 2000.
19. SUIF Compiler System, `suif.stanford.edu`.
20. T. Todman. A customisable framework for hardware compilation. PhD thesis, Imperial College, 2003.
21. T. Todman and W. Luk. Combining imperative and declarative hardware descriptions. In *Proc. 36th Hawaii Int. Conf. on System Sciences*, IEEE, 2003.
22. M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Trans. on Computer-Aided Design*, 20(2), 2001.