Incremental Elaboration for Run-Time Reconfigurable Hardware Designs

Arran Derbyshire, Tobias Becker and Wayne Luk
Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, UK
arad@doc.ic.ac.uk, tbecker@doc.ic.ac.uk, wl@doc.ic.ac.uk

ABSTRACT

We present a new technique for compiling run-time reconfigurable hardware designs. Run-time reconfigurable embedded systems can deliver promising benefits over implementations in application specific integrated circuits (ASICs) or microprocessors. These systems can often provide substantially more computational power than microprocessors and support higher flexibility than ASICs. The compilation of hardware during run time, however, can add significant run-time overhead to these systems. We introduce a novel compilation technique called incremental elaboration, which enables circuits to be dynamically generated during run time. We propose a set-based model for incremental elaboration, and explain how it can be used in the hardware compilation process. Our approach is illustrated by various designs, particulary those for pattern matching and shape-adaptive template matching.

Categories and Subject Descriptors

B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Special-purpose*

General Terms

Design, Theory

Keywords

Incremental elaboration, run-time reconfiguration, hardware compilation

1. INTRODUCTION

Over the last ten years programmable logic devices, such as Field Programmable Gate Arrays (FPGAs), have significantly increased in speed, size and density. This development has shifted the application scope of FPGAs from rapid prototyping and implementation of glue logic to the integration of full embedded systems. Modern FPGAs usually consist of a grid of configurable logic blocks in a mesh of configurable routing resources, which are controlled by static memory. It has been shown that exploiting hardware reconfigurability can significantly improve performance [16] and reduce power consumption [18].

The FPGA architecture also has the potential of updating the configuration during run time. Conceptually one part of a system can remain operational while another part is being reconfigured. Partial run-time reconfiguration can often increase the flexibility and performance of an embedded system [20], while reducing the resources required [3].

However, run-time reconfiguration involves additional design efforts of having to generate multiple configurations. These configurations can be generated either at design time or at run time. In the first case all configurations are precompiled and stored in a configuration database. At run time, these configurations are used to update the functionality of the systems. However, if hundreds or thousands of configuration files are required, the storage requirements can become a significant overhead. In this case it can be advantageous to trade off the size of the configuration storage against additional time overhead of generating configurations at run time.

It is critical to system performance to maintain a low time overhead for reconfiguration. One way to achieve this is to ensure that efficient partial reconfiguration is used. However, designing an efficient partial reconfiguration system using conventional languages and tools requires a significant increase in design effort over static design. In a reconfigurable system, the designer must provide both the structure or functionality of the hardware design, as well as the functionality to change the design at run time.

In order to reduce the design effort of creating reconfigurable systems, we introduce a new compilation scheme that generates the functionality to change a design at run time directly from a parametrised hardware description. Our approach is applicable to standard languages such as structural VHDL or Verilog. The designer can decide which parameters control the reconfiguration. This in turn has an impact on the trade-offs between run-time storage and time overheads, and the trade-offs between flexibility and performance.

A key element of our proposed technique, incremental elaboration, allows partial reconfiguration to be achieved efficiently based on run-time changes to design parameters. Overall, our approach enables a designer to create hardware designs using conventional hardware descriptions, whilst providing flexibility to the hardware structure at run time.

CASES'06, October 23–25, 2006, Seoul, Korea.

The main contributions of this paper are:

1. A compilation scheme for run-time parametrisation which contains a static and a dynamic stage, where the static stage produces incremental elaboration functions while the dynamic stage applies them (section 3).

2. A model, based on set theory, of incremental elaboration showing how it can reduce run-time design generation compared with other methods (section 4).

3. A method for making use of incremental elaboration in the hardware compilation process, using a conditional and an iteration statement as examples (section 5).

4. An evaluation of our approach based on run-time reconfigurable hardware designs, particularly for pattern matching and for shape-adaptive template matching (section 6).

2. BACKGROUND AND RELATED WORK

Although run-time reconfiguration promises compelling improvement in performance, it adds complexity to the design methods. Run-time reconfiguration requires tools to create new configurations and mechanisms to manage and load configurations at run time. In this work we shall focus on methods for developing run-time reconfigurable designs.

Parameters are used in hardware description languages to characterise the function and architecture of an implementation. A traditional use of parametrisation is the creation of library block designs, in which the parameters allow a range of functions and architectures to be generated from a single library block design. Typically, parameters are set during design time and remain static throughout run time. For reconfigurable devices, run-time parametrisation has been used to achieve dynamic specification [5].

Partial evaluation is an optimisation used in software compilation that, given a program where some of its inputs are known, yields a specialised program [7]. By applying partial evaluation to hardware descriptions, known inputs can be used to transform a hardware description into a specialised design where these inputs are constant; this is also known as constant folding. The specialised design requires fewer logic resources and is potentially faster. Partial evaluation can be used to support dynamic specialisation at run time [15].

The notion of incremental elaboration and incremental compilation has been used in both software and hardware applications. Software applications include C++ compilation [8] and elaboration of scenario-based specifications [19]. Hardware applications include elaboration for VHDL simulation [1], reconfiguration of multi-FPGA systems [9], and compilation for parallel logic verification [17]. This paper further extends this technique to support run-time reconfigurable hardware designs.

3. COMPILATION SCHEME FOR RUN-TIME PARAMETRISATION

Our approach involves a compilation scheme that can rapidly map a parametrised design to a target device at run time based on a change in its parameters.

The proposed compilation scheme employs two main compilation techniques for reducing the run time overhead. They are: 1) staged compilation and 2) incremental elaboration. Staged compilation is a compiler technique where the compilation process, after a certain stage, is postponed until run time. It is used in software compilers to perform run-time optimisations of the compiled code. Staged compilation is often split into two stages: 1) the static compilation stage and 2) the dynamic compilation stage. We use staged compilation to avoid redundantly executing compilation operations that only need to be executed once.

The main contribution of this work is the second technique, which we refer to as *incremental elaboration*. Our incremental elaboration technique uses the difference between the next values of parameters and the current values of parameters to determine the elaboration operations necessary to achieve an incremental change in the configuration of a design. It avoids the execution of operations that are redundant between parameter changes.

Our compilation scheme compiles structural hardware designs, which enables a fast compilation process. We use the following approaches to minimise the time to place logic components and route the connections between them.

- Fully-specified placement. The designer provides full placement information at compile time and describes the placement in terms of simple run-time parametrised arithmetic expressions.

- Speed-optimised routing algorithms. Routing algorithms are optimised for maximum speed of determining routes rather than minimising signal propagation delays in the routes.

Figure 1 shows the main steps in our compilation scheme. Steps 1-7 are performed in the static compilation stage and steps 8-10 are performed in the dynamic compilation stage. Steps 1-6 are the main steps typically performed in static structural hardware compilation. Steps 1-3, lexical analysis, parsing and type checking of the input source description, need only be done once since the resulting abstract syntax is not affected by a change in the parameters. Therefore these steps can be performed in the static compilation stage. Steps 4-6 are performed in the static compilation stage to generate the initial configuration of the design. The initial configuration is the first configuration used by a design at run time, which is subsequently modified incrementally when the parameters are changed. The initial configuration contains parts of a run-time parametrisable design that remain static over its entire run time.

Step 7 generates the *incremental elaboration function*, which is a function that elaborates a design incrementally according to a change in parameters. Step 7 is the key step in our compilation scheme. The incremental elaboration function is generated by syntax-directed translation which substitutes each of elaboration procedures of the input source description statements with incremental elaboration procedures. The result is a function consisting of the incremental elaboration procedures specific to the input source description. The incremental elaboration function is executed in the dynamic compilation stage.

Figure 2 shows the inputs and outputs of the elaboration function. The value of parameters of the design can change during its run time. The current parameter values are defined as the parameter values before they change, and the next parameter values are defined as the new values that the parameters change to. Using the changing parameter

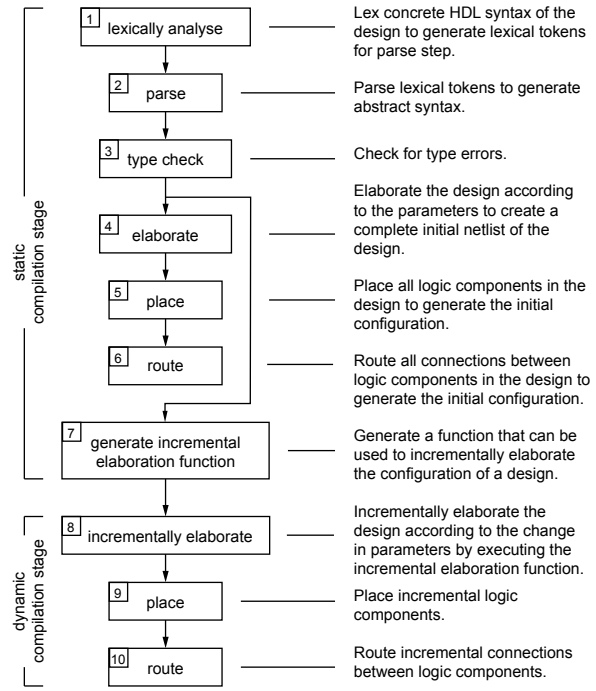

Figure 1: The key steps in our run-time parametrisation compilation scheme.

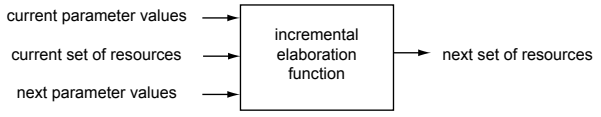

Figure 2: The inputs and outputs of the incremental elaboration function.

values, the incremental elaboration function transforms the set of resources of the design before the parameter change into a new set that implements the design according to the next values of the parameters. The set of resources of the design are the logic and routing resources that implement the design.

The incremental elaboration function is optimised by our compiler for the current design. This optimisation includes performing all incremental elaboration procedures specific to the design that only need to be executed once. The optimisation is performed in the static compilation stage. Step 7 is described in detail in sections 4 and 5.

Steps 8-10 execute the incremental elaboration function, and perform the placement and routing of the next set of resources. Steps 8-10 form the dynamic compilation stage and are repeated each time the parameters are changed. The result of steps 8-10 is a partial configuration that is used to update the configuration in the hardware, which implements the changes specified by the change in parameters. Using our compilation scheme, the incremental elaboration function and the associated place and route functions can be generated from a standalone program. They could also potentially be synthesised into hardware [23]. The resulting program, or hardware, can then be used in a run time system. This prevents the need for a run-time system to redundantly maintain the static stage of the compilation scheme.

4. INCREMENTAL ELABORATION MODEL

The generation of the incremental elaboration function is the key step in our compilation scheme (see step 7 in figure 1). This section introduces a model, based on set theory, of the elaboration of structural paramterised hardware designs. We first describe three different models of elaboration: *backtrack elaboration*, *blank elaboration* and *incremental elaboration*. We show that *incremental elaboration* has the potential of being the most efficient.

In our model the elaboration process is given by a function that elaborates the design to its logic and routing resources according to the parameters of the design. We refer to this function as the *elaboration function* of the design. Let $f_d(X_t) = Y_t$ be the elaboration function of design d, which maps the parameter set X_t with values given at time $t \in \mathbb{Z}^+$, into a set of hardware resources Y_t. The set of hardware resources Y_t represents unique logic resources in the target architecture and unique routing connections between the logic resources. We assume that there is a process that maps the statements of the structural description of the design d to the elaboration function $f_d(X_t)$ and that there is a place and route process that maps the resource set Y_t to a device configuration.

Consider the situation when parameters change at run time. Here, let Y_c be the 'current' resource set and let Y_n be the 'next' resource set. The current and next resource sets are elaborated by:

$$f_d(X_c) = Y_c \qquad \text{and} \qquad f_d(X_n) = Y_n \tag{1}$$

Let H_c be the current set of resources configured in the hardware and H_n the next set of resources. The parameter change can be achieved by removing the current resource set Y_c from the configuration and then adding the next resource set Y_n to the configuration. This can be expressed as:

$$\begin{aligned} H_n &= (H_c - f_d(X_c)) \cup f_d(X_n) \\ &= (H_c - Y_c) \cup Y_n \end{aligned} \tag{2}$$

The key characteristic of the process shown in equation 2 is that the design is completely elaborated each time the parameters are changed. Also, the current resources are completely removed from the current configuration: $H_c - Y_c$. Current resources must be removed from the current configuration so that the resources they occupy in the target architecture are free for use by the next resources.

We consider two techniques, *backtrack elaboration* and *blank elaboration*, in which current resources can be removed. In *backtrack elaboration*, the current resources are removed

by removing all the logic and routing resources according to the elaboration process that generated them. So the parameter set X_c is fully evaluated to remove the resources Y_c from H_c. Then the next parameter set X_n is evaluated and the new resources Y_n are added to $H_c - Y_c$. Backtrack elaboration is complex because the elaboration function is executed twice.

In *blank elaboration*, the current resources are removed by a function that simply blanks an area in which the current resources are situated. If the design is the only one in the configuration or if it is located in an isolated area, then it is possible to remove the configuration Y_c by resetting the hardware or deleting the reconfigurable area with a precompiled empty configuration. The next configuration H_n can then be produced by simply writing Y_n to the blank area. Blank elaboration is more efficient than backtrack elaboration because the elaboration function is only executed once. However, blank elaboration requires the run-time reconfigurable part to be spatially isolated from other resources.

In contrast to backtrack elaboration and blank elaboration, incremental elaboration aims to generate the next configuration H_n by reconfiguring only resources that change. For some parameter changes, it is possible that the resources are not completely changed and that the current and next resource sets have common elements. For these parameter changes, it may be possible to elaborate a design faster by performing an *incremental* change provided that:

$$Y_c \cap Y_n \neq \varnothing \tag{3}$$

Using a traditional compilation scheme, the resources Y_c would be removed from H_c and then the resources Y_n added to H_c. In some circumstances, it may require less operations to remove the resources $Y_c - Y_n$ from H_c and then add the resources $Y_n - Y_c$ to H_c. This would avoid any operations that involve the resources $Y_c \cap Y_n$ and so an incremental change potentially performs less operations because it reuses resources that have been elaborated previously.

We refer to the process of elaborating the resources incrementally from a change in parameters as *incremental elaboration*. We define the sets $R = Y_c - Y_n$ and $S = Y_n - Y_c$, and refer to them as the *incremental sets*. Incremental elaboration is illustrated by a Venn diagram in figure 3.

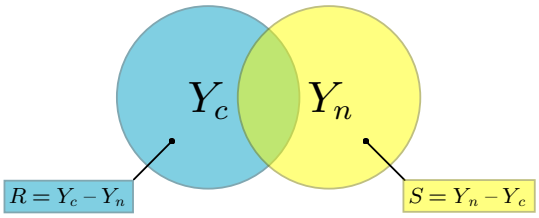

Figure 3: A change in resources from the current resource set Y_c to the next resource set Y_n, from a change in parameters X_c to X_n. The incremental sets $R = Y_c - Y_n$ and $S = Y_n - Y_c$ represent the resources that are elaborated by incremental elaboration in order to achieve the change in parameters without needing to elaborate the resources $Y_c \cap Y_n$.

This process is similar to the technique of finding a partial configuration [14]; the distinction is that we are interested in the source statements that elaborate to these sets. This is because, if the source statements can be manipulated in such a way so that parameter changes only elaborate to the incremental sets, then it is possible to incrementally elaborate a design. This gives the potential to be able to change the resources of the design faster, since it usually takes longer to elaborate a larger set.

We define an *incremental elaboration function* to be one that elaborates the resources of design d incrementally:

$$g_d(X_c, X_n, H_c) = H_n \tag{4}$$

The incremental sets can be expressed as:

$$R = f_d(X_c) - f_d(X_n) \qquad \text{and} \qquad S = f_d(X_n) - f_d(X_c) \tag{5}$$

H_n can now be produced by removing R from H_c and adding S:

$$H_n = (H_c - R) \cup S \tag{6}$$

However, we do not want to determine the sets R and S as described in equation 5 because this would require a complete evaluation of $f_d(X_n)$ and $f_d(X_c)$. In this case no operation would be saved compared to backtrack elaboration. Instead, our objective is to compile the source statements of a design to a function $g_d(X_c, X_n, H_c)$ such that for all parameter changes:

1. the incremental elaboration function can improve the speed of run-time elaboration: the time to execute $H_n = g_d(X_c, X_n, H_c)$ is less than or equal to the time to execute $H_n = (H_c - f_d(X_c)) \cup f_d(X_n)$,
2. operations that elaborate to the set $Y_c \cap Y_n$ are minimised,
3. the incremental sets are minimal; $R \cap S = \varnothing$ is true,
4. the result is correct; $g_d(X_c, X_n, H_c) = [H_c - (f_d(X_c) - f_d(X_n))] \cup [f_d(X_n) - f_d(X_c)]$ is true.

For these given objectives, incremental elaboration should always be more effective than backtrack elaboration because it does not fully evaluate $f_d(X_n)$ and $f_d(X_c)$. In comparison to blank elaboration, incremental elaboration has the potential of being more efficient. However, the actual benefit could vary depending on the number and type of operations to be executed. We illustrate this in detail in section 6.

5. COMPILING INCREMENTAL ELABORATION FUNCTIONS

Based on the model presented in the last section, we introduce our compilation technique for compiling incremental elaboration functions and describe how it satisfies our objectives. We show how the basic control statements of a hardware description language can be compiled to an incremental elaboration function. From the compilation of basic control statements, general methods can be derived for compiling a hardware description to its incremental elaboration function.

The incremental sets can be found by evaluating the difference between $f_d(X_c)$ and $f_d(X_n)$. The challenge is that the values of the run-time parameters in X_c and X_n are not known at compile time. One solution is to determine how the parameters may change at run time, and for each of these parameter changes, use this information to evaluate the incremental sets. We show how this solution works for two basic parametrisable control statements of a standard hardware description language.

Consider the following design `cond` that consists only of a single conditional statement. Figure 4 shows a schematic description of the example. The example could also be easily described in a structural hardware description.

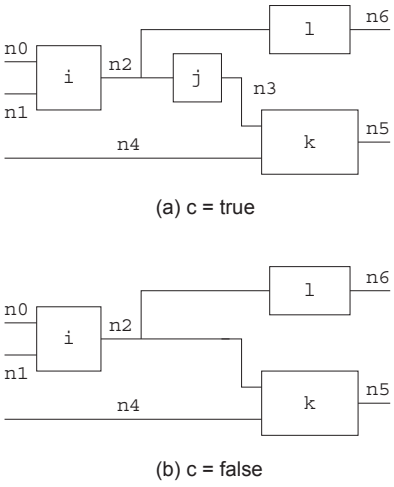

Figure 4: Schematic description of the example `cond` when (a) parameter c = true and when (b) parameter c = false.

An appropriate elaboration function for this example is shown in equation 7. The elaboration function is generated from the design description by including an equivalent conditional statement. To elaborate the design, the parameter c is tested to see if it is true, if so then f_{cond} returns the set A otherwise it returns the set B. The set A represents the instances `i`, `j`, `k` and `l` and the nets `n0 - n6`. The set B represents the instances `i`, `k` and `l` and the nets `n0 - n2` and `n4 - n6`. The set D is used as a temporary variable. After the execution of f_{cond}, the resulting set of resources D would be mapped into hardware resources.

$$
\begin{array}{l}
f_{cond}(c) = \\
\quad \textbf{if } (c) \textbf{ then} \\
\quad\quad D := A \\
\quad \textbf{else} \\
\quad\quad D := B \\
\quad \textbf{return } D
\end{array}
\tag{7}
$$

At run time, each parameter can change in a number of ways that is determined by the parameter type. Here, c changes in two cases only, when 1) the current value is false and the next value is true, and 2) the current value is true and the next value is false. It is necessary to check for these two conditions only, and then we can evaluate the incremental sets. The key step for producing the incremental elaboration function g_{cond} is to add conditional statements that check for these parameter changes.

We refer to these added conditional statements as *Δ-conditionals*. A Δ-conditional is defined as a statement $\delta \in \Delta$, where Δ is a set of conditional statements derived from each control statement of the source description. The Δ-conditionals replace the definitions of R and S in g_{cond} for the example as shown in equation 8.

$$
\begin{array}{l}
g_{cond}(c_c, c_n, H_c) = \\
\quad \textbf{if } (c_c \wedge \neg c_n) \textbf{ then} \\
\quad\quad R := f_{cond}(c_c) - f_{cond}(c_n) \\
\quad\quad S := f_{cond}(c_n) - f_{cond}(c_c) \\
\\
\quad \textbf{if } (\neg c_c \wedge c_n) \textbf{ then} \\
\quad\quad R := f_{cond}(c_c) - f_{cond}(c_n) \\
\quad\quad S := f_{cond}(c_n) - f_{cond}(c_c) \\
\quad \textbf{return } (H_c - R) \cup S
\end{array}
\tag{8}
$$

The next step is to evaluate the sets R and S. The parameter change conditions make this possible. If the condition of a Δ-conditional is true, then the current and next values of the parameters for the body of that Δ-conditional can be determined. In the case of a conditional statement, the exact parameter values can be determined from the Boolean expressions of the conditions. Therefore, it is possible to apply the value within each condition and then evaluate each occurrence of f_{cond} according to equation 7.

$$
\begin{array}{l}
g_{cond}(c_c, c_n, H_c) = \\
\quad \textbf{if } (c_c \wedge \neg c_n) \textbf{ then} \\
\quad\quad R := A - B \\
\quad\quad S := B - A \\
\quad \textbf{if } (\neg c_c \wedge c_n) \textbf{ then} \\
\quad\quad R := B - A \\
\quad\quad S := A - B \\
\quad \textbf{return } (H_c - R) \cup S
\end{array}
\tag{9}
$$

Shown in equation 9 is a general result for a simple conditional statement with two alternatives. The next step is to find the difference between the sets A and B:

$$
\begin{array}{l}
g_{cond}(c_c, c_n, H_c) = \\
\quad \textbf{if } (c_c \wedge \neg c_n) \textbf{ then} \\
\quad\quad R := \{j, (n2, \{i, j, l\}), (n3, \{j, k\})\} \\
\quad\quad S := \{(n2, \{i, k, l\})\} \\
\quad \textbf{if } (\neg c_c \wedge c_n) \textbf{ then} \\
\quad\quad R := \{(n2, \{i, k, l\})\} \\
\quad\quad S := \{j, (n2, \{i, j, l\}), (n3, \{j, k\})\} \\
\quad \textbf{return } (H_c - R) \cup S
\end{array}
\tag{10}
$$

Note that net set elements are represented uniquely as a tuple with their identifier and the instances they connect to.

The second basic control statement is a single iterative statement. In the following we show how an iterative statement with a parametrised upper bound and a fixed lower bound can be compiled into an incremental elaboration function.

$$
\begin{array}{l}
f_{iter}(c) = \\
\quad\quad D := A \times I_0^c \\
\quad\quad \textbf{return } D
\end{array}
\tag{11}
$$

The iterative statement is modelled by the Cartesian product of A and the set I. The purpose of I is to provide a unique element for each iteration of A in terms of the iteration index. The set I with a lower bound u and upper bound v is defined as:

$$I_u^v = \{i \mid i \in \mathbb{Z} \wedge u \leq i \leq v\} \tag{12}$$

Now, the Δ-conditionals of the iterative statement with the upper bound determined by c must be found. Again, the aim is to evaluate the incremental sets R and S at compile time, without elaborating the resources $Y_c \cap Y_n$ and without knowing the exact values of c_c and c_n. Irrespective of its value, c can: 1) increase, and 2) decrease. This leads to the Δ-conditionals $c_n > c_c$ and $c_n < c_c$. The incremental evaluation function g_{iter} can then be expressed as shown in equation 13.

$$\begin{array}{l} g_{iter}(c_c, c_n, H_c) = \\ \quad \textbf{if } (c_n > c_c) \textbf{ then} \\ \quad\quad R := \varnothing \\ \quad\quad S := A \times I_{c_c+1}^{c_n} \\ \quad \textbf{if } (c_n < c_c) \textbf{ then} \\ \quad\quad R := A \times I_{c_n+1}^{c_c} \\ \quad\quad S := \varnothing \\ \quad \textbf{return } (H_c - R) \cup S \end{array} \tag{13}$$

The function can be explained in terms of the ways in which the upper bound parameter changes. When c increases, no resources are removed and the resources A, iterated between $c_c + 1$ and c_n, are added. When c decreases, the opposite results. The expressions show that it is not necessary to recompile the whole iteration. Only the top part of the iteration needs to be changed, which gives an efficient way to compute the incremental sets.

For combined control statements, the Δ-conditionals have to be nested with each other for all possible combinations of parameter changes. With the two basic control statements and combination of these, it is possible to cover the basic features of a paramterised structural hardware description. It is also straightforward to extend the model to cover other language features, for example the parametrised placement of instances or iterative statements with parametrised lower bounds.

We propose a general method to produce $g_d(X_c, X_n, H_c)$ by applying *partial evaluation* [7] to $g_d(X_c, X_n, H_c)$ using the values of parameters determined by the conditions in which they change. The key steps are:

1. generate Δ-conditionals
2. apply partial evaluation to the $f_d(X_c)$ and $f_d(X_n)$ terms
3. evaluate the differences between sets

For the single conditional statement example in equation 7, step 1) corresponds to equation 8, step 2) to equation 9 and step 3) to equation 10.

Our compiler is based on the above steps. The compiler finds parameters and builds up the environments c and n while generating the incremental elaboration function. For each parameter a new environment is created for every Δ-conditional. The algorithm works recursively on the list of source statements and produces a list of statements that form the incremental elaboration function. The resulting function is then further reduced and optimised to reduce function size and execution time for run-time configuration generation.

6. EXPERIMENTAL RESULTS

To determine the effectiveness of our compilation method, we compare measurements of the execution time of incremental elaboration functions produced by our compiler with the execution time of other elaboration approaches.

The performance improvement of incremental elaboration in comparison to backtrack elaboration or blank elaboration can depend on many possible parameter changes. Each parameter change can involve the execution of three different kind of operations: 1) elaborating control statements that include conditional statements, Δ-conditional statements and iterative statements, 2) elaborating the elements in the logic resource sets and 3) elaborating the elements in the routing resource sets. A complete analysis would involve counting the number of operations for a given parameter change and applying cost factors according to the complexity of each operation. For an abstract analysis it is difficult to apply appropriate cost factors to abstract operations. However, it is safe to assume that routing operations are most complex, independent from specific target architectures. Hence, if a parameter change involves a reasonable amount of routing operations, then the number of routing operations can be used as an estimate for comparing the efficiency of different algorithms. We shall look at two designs in detail, then we discuss design trade-offs that our compiler can help to explore.

6.1 Pattern Matcher

The first case study is an architecture-independent analysis of a pattern matcher. This is a simple example to illustrate some of the types of specialisation that can be achieved by run-time parametrisation. The design matches a variable length sequence of bits, or 'pattern', against the bits of a stream of data. A further sequence of bits, or 'mask', determines which bits of the pattern are active in the matching process. The mask allows different length patterns, and separated fragments of patterns, to be matched.

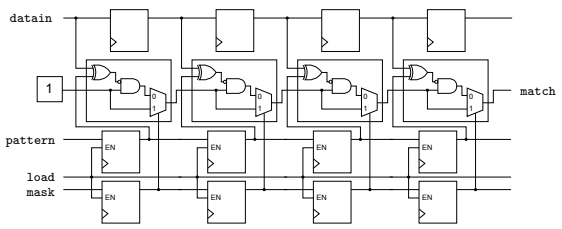

Figure 5: Static implementation of the pattern matcher.

Figure 5 shows a static implementation of the pattern matcher design. This implementation has shift registers that hold the pattern, mask and a section of the input data

stream. The logic compares the pattern with the data according to the mask. By representing the pattern and mask as parameters, it is possible to simplify the logic by treating the pattern and mask signals as constants and applying Boolean simplifications.

The pattern matcher can be parametrised with the parameters `m`, `pattern` and `mask`, where `m` is the number of match elements. A run-time parametrised version can be described by an iterative statement with upper bound `m` to replicate the match element. The parametrisation of the pattern and mask can be described by nesting two conditional statements within the iterative statement. Figure 6 shows an implementation of the run-time parametrised pattern matcher with the parameters `m=4`, `pattern = [0, 1, x, 0]` and `mask = [1, 1, 0, 1]`.

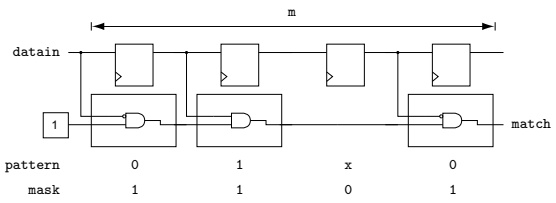

Figure 6: Run-time parametrisable implementation of the pattern matcher.

The pattern matcher is implemented in RTPebble [3]. Pebble [10] is a structural hardware description language and it features are subset of VHDL. RTPebble provides extensions to model run-time parameters. These extensions can be applied to the industry standard languages VHDL or Verilog. The parametrised RTPebble description of the pattern matcher is implemented with our compiler tools.

In our case study we modify the parameter `m` to allow different length patterns to be matched whilst maintaining the minimum resource usage. Changing `m` involves adding and removing both routing and logic resources. Figure 7 shows, for different previous values and decrements of the parameter m, how the number of routing operations executed by incremental elaboration compares to the number executed by (a) backtrack elaboration and (b) blank elaboration. For both graphs in figure 7, the y axis shows the number of routing operations that are saved by incremental elaboration over backtrack elaboration, the x axis shows the amount by which the parameter is decreased, and the z axis shows the previous parameter value. The greatest saving in routing operations occurs for large parameter values and for a small parameter decrement. For the values shown in figure 7(a), incremental elaboration always executes less routing operations than backtrack elaboration to decrease an iterative statement parameter value. The graphs suggest that this trend would continue for higher parameter values and it seems likely that incremental elaboration will, for all parameter values, execute either an equal or lower number of routing operations to backtrack elaboration with decreasing parameter values.

In contrast to backtrack elaboration, it is clear from figure 7(b) that, for some of the previous parameter and decrement value combinations, blank elaboration uses fewer operations than incremental elaboration. These combinations are shown on the graph where the number of routing operations has a negative value (the top of the bars are black for negative values). Therefore, for the pattern matcher it is possible to make a decision, depending on the change in parameters, as to whether incremental elaboration or blank elaboration should be used, in order to optimise the number of routing operations.

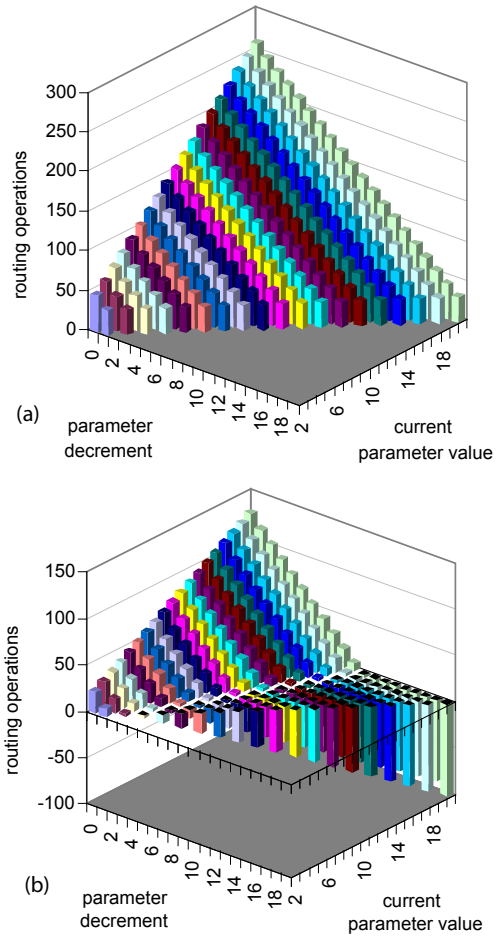

Figure 7: Graphs showing the saving in routing operations of the incremental elaboration function of the pattern matcher design over (a) backtrack elaboration, and (b) blank elaboration, when the parameter m is decreased. Where the tops of the bars are black, the bars have a negative value.

6.2 SA-TM Video Search Processor

The architecture specific case study is a Shape-Adaptive Template Matching (SA-TM) video search processor. This design is derived from the SA-TM method for retrieving arbitrarily shaped objects from video streams [4]. The SA-TM method is used to search for a template image within a video

stream by calculating the similarity between the template pixels and the video image pixels. To enable the template image to have an arbitrary shape, a mask image is used to define the shape of the template image. The mask image performs a similar function as the mask of the pattern matcher described previously. In this analysis, the template and mask parameters become run-time parameters. For arbitrary template and mask images, it is impractical to precompile all specialised configurations at design time because of the exceedingly large number of possible combinations. Thus, an SA-TM processor is an interesting case study for run-time configuration generation.

The analysis is performed on a Xilinx Virtex XCV1000 FPGA on a Celoxica RC-1000 PCI card in a host PC. The design description in RTPebble is compiled to a Java program which consists of the elaboration and incremental elaboration functions with method calls to the JBits API [6]. The run-time controllers, including the incremental elaboration functions, are written directly in Java and executed on the host PC. To reconfigure the design, the current configuration is sent to the incremental elaboration function with new parameter values to produce the next configuration.

The following describes the trade-offs between the peak throughput and area of static and run-time parametrisable SA-TM designs. Table 1 shows the peak throughput and area trade-offs of the SA-TM designs with their configuration update time. All designs have a template image of 12×12 pixels, a mask image of 12×12 pixels, and process a search image of 100×100 pixels. In our experiments the template image is limited to 12×12 pixels because a simple layout is used which restricts the image size to the height of the FPGA. Larger images can be supported simply by changing the layout to use resources across the device, and by using larger FPGAs now available.

For the static designs, the configuration update time is the time to place and route the designs with the standard Xilinx tools from a netlist that has already been mapped to the Virtex device. For the run-time parametrisable designs, the configuration update time is the time to generate a configuration based on a change in the template and download the configuration to the FPGA.

The static design using distributed RAM is a direct implementation of the SA-TM algorithm. This design provides the lowest throughput and requires the largest area. However, the static version of the design can be improved by using shift registers. This version has a higher peak throughput while using less area. The run-time parameterisable version of the design is 57% smaller and provides less performance. With additional pipelining, the run-time parameterisable design is 41% smaller than the static version and achieves higher peak throughput.

The configuration update time for a template change is significantly lower compared to the standard place and route tool. It can also be seen from table 1, that the incremental elaboration function compiled with both template and mask as run-time parameters (a) takes longer to evaluate a template parameter change than the incremental elaboration function compiled for template as the only run-time parameter (b). This is because the incremental elaboration function is more complex for the combined control statements of the two parameters. In case (b), the configuration update time is almost the same as the time for downloading a configuration from a database, which can be explained by the fact that no routing performed for a template change, only changes to logic look-up tables are performed. A mask change, on the other hand, requires routing. Other experiments show that for a 50% change to the mask, incremental elaboration with routing takes around 2s, or 2000ms, and for a 100% change to the mask takes around 4s, or 4000ms, which is still faster than the complete blank elaboration by the standard place and route tool.

The run-time system described above is intended for exploring and experimenting with design methods and tools, and requires a host PC environment and the JBits API. The incorporation of the Java Virtual Machine (JVM) in our compilation process imposes addition overhead of time and computational resources to generate new configurations. Also to this date, there is no JBits support available for the latest Virtex-4 architecture. For these reasons, we are migrating the run-time system to a new database and onto the FPGA itself. Recent work has shown a growing trend for adaptive embedded systems employing run-time self-reconfiguration [2], [12], [13].

It has been proposed that a run-time system can be directly implemented on the FPGA based on an embedded processor and an internal configuration port [2]. Embedded processors, such as the Xilinx MicroBlaze softcore [21] or PowerPC hardcore processor [22], provide a capable means of embedded processing for FPGAs. In combination with the Internal Configuration Access Port (ICAP) [2], they can form systems that quickly modify themselves. These systems usually employ module-based reconfiguration that loads pre-compiled configurations into a dedicated module area on the FPGA. The old module is completely overwritten by the new configuration.

Recent work suggests a new method of merging configurations without fully overwriting the target area [13], and an algorithmic approach for run-time configuration generation [11]. However, the proposed designs often involve special-purpose run-time systems, and are not capable of compiling arbitrary configurations. We intend to develop an embedded version of our run-time system that allows efficient run-time compilation of parametrised hardware descriptions. Even though it is conceivable to run the JVM on an embedded processor, we seek to develop our run-time system based on a more efficient and native representation of the configuration database.

6.3 Design Trade-Offs

To provide an architecture independent measure of the complexity of incremental elaboration functions, we quantify how the parametrisation of different designs affects their size. The size of an incremental elaboration function indicates the design effort that is saved for the designer by using our compiler over manually producing this function. The size of the function also correlates directly to the memory requirements of the reconfiguration controller. The number of statements in the elaboration function is obtained from the source code of the design. We compare this against the number of statements in the incremental elaboration function which is the result of our compilation.

Table 2 shows the elaboration function sizes for the pattern matcher in section 6.1 and the SA-TM design in section 6.2. Column C gives the count of the control operations, column L the count of the logic operations and R the routing operations.

SA-TM Design		Configuration Generation	Peak Throughput (Mpixel/s)	Area (slices)	Configuration Update Time (ms)	
static	distributed RAM	blank elaboration (Xilinx P&R)	25	4049	286000	
	shift register	blank elaboration (Xilinx P&R)	44	3578	255000	
					(a)	(b)
RTP	shift register	incremental elaboration	38	1539	42	0.75
	shift register, pipelined	incremental elaboration	59	2115	42	0.75
	shift register	configuration database	38	1539	0.73	
	shift register, pipelined	configuration database	59	2115	0.73	

Table 1: Speed, area and configuration update time trade-offs for SA-TM designs implemented in a Virtex FPGA. The configuration update time is the time to generate a configuration from a change in the template parameter and download the configuration to the FPGA. In (a) the incremental elaboration function is compiled with both template and mask as run-time parameters, and in (b) the incremental elaboration function is compiled with only template as a run-time parameter.

Case Study	Parameters		Elaboration Function Statements				Incremental Elaboration Function Statements			
	Static	Dynamic	C	L	R	Total	C	L	R	Total
Pattern Matcher	m	pattern	2	8	14	24	3	4	0	7
		m, pattern	2	8	14	24	18	24	28	70
	m	pattern, mask	3	10	19	32	10	16	20	46
		m, pattern, mask	3	10	19	32	62	104	168	334
SA-TM	w, h, c	temp., mask	10	19	45	74	14	42	108	164
	w, h	c, temp., mask	10	19	45	74	68	200	536	804
	c	w, h, temp., mask	10	19	45	74	492	1602	4500	6594
		w, h, c, temp., mask	10	19	45	74	1956	6140	17356	25452

Table 2: Size of elaboration and incremental elaboration functions for case studies.

It is obvious that the number of source statements is constant over all of the static and dynamic parameter combinations for each design. On the other hand, the number of incremental elaboration function statements increases rapidly to cover different combinations of configuration possibilities as the number of dynamic parameters increases. This is because the size of the incremental elaboration function depends directly on the number of nested Δ-conditionals, given by the number of dynamic parameters. However, incremental elaboration is much faster than full elaboration as shown in section 6.2. This shows that we can trade off the degree of dynamic parametrisation against the memory requirements of the reconfiguration controller. Such trade-off can be obtained rapidly by our compiler, which provides an effective and easy means of exploring the design space of run-time reconfigurable applications.

7. SUMMARY

This paper presents a novel compilation scheme for efficient generation of hardware configurations. It compares three different elaboration methods, with special focus on incremental elaboration which aims to generate new configurations directly from parameter changes. We describe the compilation process of incremental elaboration functions, and illustrate a run-time system for rapid configuration generation and its application in producing various designs. Current and future work includes extending our incremental compilation approach to cover systems with both hardware and software components, and for a wide range of applications.

Acknowledgements

The support of Celoxica, Xilinx and the UK Engineering and Physical Sciences Research Council is gratefully acknowledged.

8. REFERENCES

[1] T. Ahn, K.H. Kim, S. Park, and K. Choi, "Incremental Analysis and Elaboration of VHDL Description". In *Proc. Third Asia Pacific Conf. on Hardware Description Languages*, pp. 128–131, 1996.

[2] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. "A Self-Reconfiguring Platform". In *Field-Programmable Logic and Applications*, LNCS 2778, pp. 565–574. Springer, 2003.

[3] A. Derbyshire and W. Luk. "Compiling Run-Time Parametrisable Designs". In *Proc. IEEE Int. Conf. on Field-Programmable Technology*, pp. 44–51. IEEE, 2002.

[4] J. Gause, P.Y.K. Cheung, and W. Luk. "Reconfigurable Shape-Adaptive Template Matching Architectures". In *Proc. IEEE Int. Symp. on Field-Programmable Custom Computing Machines*, pp. 98–107. IEEE Computer Society Press, 2002.

[5] S. Guccione and D. Levi. "Run-Time Parameterizable Cores". In *Field-Programmable Logic and Applications*, LNCS 1673, pp. 215–222. Springer, 1999.

[6] S. Guccione, D. Levi, and P. Sundararajan. "JBits: Java Based Interface for Reconfigurable Computing". In *Proc. Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*. The John Hopkins University, 1999.

[7] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[8] M. Karasick. "The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler". *ACM SIGSOFT Software Engineering Notes*, vol. 23, pp. 131–142, 1998.

[9] K.K. Lee and D.F. Wong. "Incremental Reconfiguration of Multi-FPGA Systems". *Proc. ACM Int. Symp. on Field Programmable Gate Arrays*, ACM, pp. 206–213, 2002.

[10] W. Luk and S. McKeever. "Pebble: A Language for Parametrised and Reconfigurable Hardware Design". In *Field-Programmable Logic and Applications*, LNCS 1482, pp. 462–472. Springer, 1998.

[11] P. Lysaght and D. Levi. "Of Gates and Wires". In *Proc. 18th International Parallel and Distributed Processing Symposium*, pp. 132–137. IEEE Computer Society Press, 2004.

[12] P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, and W. Luk. "A Reconfigurable Platform for Real-Time Embedded Video Image Processing". In *Field-Programmable Logic and Applications*, LNCS 2778, pp. 606–615. Springer, 2003.

[13] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. "Module dynamic reconfiguration in Virtex FPGAs". *IEE Proc. Computers and Digital Techniques*, 153(3):157–164, 2006.

[14] N. Shirazi, W. Luk, and P.Y.K. Cheung. "Framework and Tools for Run-Time Reconfigurable Designs". *IEE Proc. Computers and Digital Techniques*, 147(3):147–152, 2000.

[15] S. Singh, J. Hogg, and D. McAuley. "Expressing Dynamic Reconfiguration By Partial Evaluation". In *Field-Programmable Custom Computing Machines*, pp. 188–194. IEEE Computer Society Press, 1996.

[16] H. Styles and W. Luk, "Compilation and management of phase-optimized reconfigurable systems". *Proc. Int. Conf. on Field Prog. Logic and Applications*, pp. 311–316, 2005.

[17] R. Tessier and S. Jana, "Incremental Compilation for Parallel Logic Verification Systems". *IEEE Trans. on VLSI Systems*, 10(5):623–636, October 2002.

[18] R. Tessier, S. Swaminathan, R. Ramaswamy, D. Goeckel, and W. Burleson. "A Reconfigurable, Power-Efficient Adaptive Viterbi Decoder". *IEEE Transactions on VLSI Systems*, 13(4):484–488, April 2005.

[19] S. Uchitel, J. Kramer, and J. Magee. Incremental Elaboration of Scenario-based Specifications and Behavior Models using Implied Scenarios". *ACM Trans. on Software Engineering and Methodology*, 13(1):37–85, January 2004.

[20] M.J. Wirthlin and B.L. Hutchings, "Improving Functional Density using Run-Time Circuit Reconfiguration," *IEEE Trans. on VLSI Systems*, 6(2):247–256, June 1998.

[21] Xilinx. *MicroBlaze Microcontroller Reference Design User Guide v1.5*. September 12, 2005.

[22] Xilinx. *PowerPC 405 Processor Block Reference Guide*. July 20, 2005.

[23] S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, and D. Levi. "A High I/O Reconfigurable Crossbar Switch". In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 3–10. IEEE Computer Society Press, 2003.