High-level Language Extensions for Run-time Reconfigurable Systems

T.K. Lee, A. Derbyshire and W. Luk
Department of Computing
Imperial College London
{tkl97, arad, wl}@doc.ic.ac.uk

P.Y.K. Cheung
Department of Electrical Engineering
Imperial College London
p.cheung@imperial.ac.uk

Abstract

This paper presents high-level language extensions for designs that can be reconfigured at run time. Such extensions provide a unified framework for instantiating and controlling reconfigurable hardware blocks. Our framework involves capturing functional blocks at the task level, with language constructs for describing run-time reconfigurable tasks, and dynamic datatypes for describing run-time parametrisable designs. Two compilation paths, one involving the Handel-C system and the other involving the RT Pebble tools, have been developed. The effectiveness of our approach has been evaluated using designs for shape-adaptive template matching and network firewall.

1 Introduction

Dynamic optimisation based on hardware specialisation at run time can improve performance and reduce resource usage [9]. This technique has been shown to benefit many applications, such as cryptography [7], image processing [4] and networking [11].

Hardware design tools have been developed to facilitate exploitation of run-time reconfiguration. For instance, the Lava system uses a client-server architecture to enable rapid generation of configuration bitstreams remotely [9]. The Lava language provides primitive components to allow circuits to be described as blocks. Such hardware blocks can be instanced and placed. Lava, however, is intended for producing high-quality structural designs. Another system, JHDL [1], provides an object-oriented model to manage reconfigurable resources. Circuits are developed in an object hierarchy. JHDL supports run-time and partial configuration. A single description can integrate both the software simulation and hardware execution.

Both Lava and JHDL are fine for capturing reconfiguration at the hardware-logic level of abstraction. This paper proposes an approach complementary to their work: adopting language extensions that enable the description of run-time reconfigurable designs and their run-time control in a single representation. This approach should also provide us an abstraction that facilitates re-use of run-time parametrisable designs [5].

The contributions described in this paper include:

- unified language extensions for specifying run-time parametrisable designs, run-time reconfiguration and the corresponding run-time control;
- two compilation paths that support the proposed language for designs with and without run-time reconfiguration;
- case studies involving run-time parametrisable designs for video searching and for network firewall using our high-level description language.

2 Framework overview

Figure 1 shows an overview of our framework for developing run-time reconfigurable designs with a high-level description language. There are four design objectives: (1) to provide abstractions for systems that involve hardware software co-operation; (2) to allow run-time reconfiguration to be captured explicitly in a high-level language; (3) to support the inclusion of dynamic components, such as run-time parametrisable cores, in a design; (4) to provide a unified description for the instantiation and run-time control of reconfigurable hardware blocks.

Our approach uses a single uniform C-like description, RT C, to accomplish the task of specifying hardware designs and software host controls. We adopt a simple model in which functional operations, for both the hardware and the software, are described in task blocks. This feature reduces the effort required in performing manual hardware software partitioning.

In addition, the same description can be used to specify both a static functional design and its run-time behaviour. Specialised constructs are provided for scheduling run-time reconfiguration. Run-time parametrisation is supported through the introduction of a run-time parameter

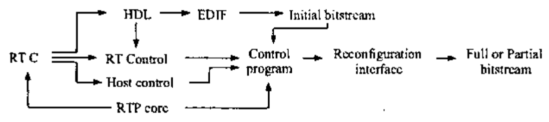

Figure 1: Design flow of our framework.

type. The ability to acquire run-time parametrisable cores is achieved through the use of an interface.

A top-level host control facility is introduced for coordinating hardware and software operations, monitoring run-time reconfiguration, and updating a design according to the run-time parameters. Optimisation control through the use of constraint expressions is also supported.

The design flow contains two paths: (1) The hardware part of a design is extracted from the high-level description and will then be translated to a low-level hardware description language (HDL). This low-level HDL will be used to produce an initial bitstream using main-stream hardware synthesis tools. (2) The software host control and the run-time control information are deduced from the specification requirements and will then be used to create a control program. Device specific reconfiguration interface will be used to generate partial bitstreams if desired.

Our focus is on run-time reconfiguration language extensions and their associated run-time control requirements. We follow a simple language model. The language supports C-style operations on base data type with an arbitrary data width size. In addition, it allows specifying operations that make use of hardware performance, such as parallel processing.

We treat Handel-C as a language to be extended. We test our designs through a modified version of a low-level hardware description language Pebble [3], the Handel-C language and the run-time reconfiguration interface JBits.

3 Language extensions

This section discusses the issues in extending a high-level language to describe run-time reconfiguration. The language should allow designers to explicitly specify the reconfiguration requirements. At the same time, it should provide abstraction from tedious low-level control mechanisms. The purpose is to enable designers to control reconfiguration at a high level, while freeing them from low-level considerations.

Our high-level language unifies hardware design and software host control in a single description. Functional operations are structured into task blocks. There are three types of tasks. Run-time reconfigurable tasks are those specified in a reconfigurable construct. Run-time parametrisable tasks are those that contain one or more dynamic data type. Static tasks are those that do not fall into the previous two categories.

We describe a task as the smallest unit of a hardware functional block that can be reconfigured. It would be possible to allow reconfiguration down to the expression or data item levels. However, every reconfiguration incurs an overhead of reconfiguration time and keeping track of the resource usage. Therefore, it is a trade-off between the efficiency in manipulating a larger hardware block and the flexibility of identifying a smaller logic element.

A task is basically similar to a C function but with several additional restrictions. A task has no return type and does not return any value. There is also no global variable allowed within a task description. These restrictions are required, in order to facilitate an efficient hardware software partitioning, and optimisation in processing of several tasks in parallel.

To allow a task to communicate with another task or with a software host, the argument list is used. Data flow into and out of a task is specified in the argument list. There are two classes of argument: either *in only*, or *in and out*. The former class allowes a task to receive data. While the latter is mainly for sending data, it can also be used as a data input similar to a pointer to a memory location in standard C. Although the communications are synchronous in nature, buffering is still employed to allow asynchronous parallel processing. On-board SRAMs are used to buffer large quantity of data such as a video image, while on-chip registers are used for small amount of data.

A task is started when the host processing encounters a reference of the task in a design description. A task is terminated, when it is out of an execution scope, for instance, at the end of each iteration of a `for` loop.

A task is described by using the `TASK` functional class specifier. To indicate that a data type is dedicated for *in and out*, an `&` sign is used after the data type specifiers in an argument list. Figure 2 shows an example task description.

Unlike a C function, which is shared by every instance that it is used, a task is not shared by default. Each use of a task in different parts of a design description will instantiate a different instance of the corresponding hardware block. However, repeated use of the same task, such as in a loop, will not instantiate another instance. By not sharing an instance of a task, it is possible to allow parallel processing on different instances. To allow a task to be shared, the `shared` qualifier can be used before the `TASK` keyword.

Each instance of a task is translated into a corresponding hardware block in the low-level language. Notice that any statement that is not in a task block can be taken as software execution code in the host control.

```
TASK taskid(int 8 arg1, int& 8 arg2) {
        // do something
    }
```
(a) An example task description

arg1
taskid
arg2

(b) A block-level view of (a)

Figure 2: This example describes the task *taskid*, which has two arguments. The first argument *arg1* is of type `int` and is used as data in. The second argument *arg2* is also of type `int` and is used as data out. Both these two arguments have a data width size of 8 bits.

```
RECONF sync  taskarrid[arrsize];
```
(a) An example reconfigurable task array

```
RECONF sync struct  taskstructid {
                       task0id(...);
                         ...
                       taskNid(...);
                    };
```
(b) An example reconfigurable task struct

```
RECONF sync union  taskunionid {
                      task0id(...);
                        ...
                      taskNid(...);
                   };
```
(c) An example reconfigurable task union

Figure 3: Examples of reconfiguration construct.

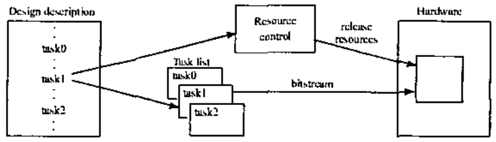

Figure 4: An illustration of task reconfiguration. When a host processing encounters a reference to *task1* in the design description at run time, it initiates a reconfiguration. The corresponding bitstream will then be uploaded to the hardware.

4 Reconfiguration constructs

To allow designers to explicitly specify reconfiguration, we introduce three different classes of reconfiguration construct. Each class provides different types of performance consideration and flexibility in specifying reconfiguration. A designer has control over which part of a design is to be reconfigured, when the reconfiguration will occur, and how the reconfiguration is to be carried out. These constructs are characterised by the specifier RECONF. Figure 3 shows some examples of reconfiguration construct. Figure 4 illustrates a task reconfiguration.

All the tasks within the same construct are supposed to share virtual hardware over time. The resources concealed by a particular task will be released, if a reconfiguration of another task from the same construct is required. When a description encounters a reference to a task that is not already in hardware, a run-time reconfiguration will be initiated. Hence, a designer can explicitly specify the scheduling of task reconfiguration in a design description.

A task assigned as the first element of a reconfiguration construct is considered as the default task. A default task will form part of the initial bitstream, and will act as a place holder on the hardware for all the tasks in the same construct. When a reconfiguration is specified, the resources concealed by the default task will be released and displaced by the upcoming task. Since a default task is already in the initial bitstream, it therefore can be used immediately without reconfiguration delay, provided that the task has not been displaced.

The optional `sync` type qualifier requests a synchronization to be performed at both the beginning and the end of a task. A blocking communication is conducted between a hardware task and the software host. Each time before a task is started, it waits until the software host has sent a synchronization signal. Similarly, after the task has finished its processing, it will also send a synchronization signal to the blocked software host.

Tasks that are not assigned to any one of these three constructs imply that they are static, except those involving the dynamic data type that will be discussed later in this section. Static tasks will not be subjected to reconfiguration.

A reconfiguration construct will be translated to a list of hardware blocks in the low-level HDL. Appropriate annotations, which indicates reconfiguration information, will be added to the HDL description. A separate description of the corresponding run-time requirements is also produced for the host control.

Reconfigurable task array. This construct, see Figure 3(a), provides a syntactically convenient way to specify a list of tasks. A parameter can be set to a default value, so that it will behave as a task struct or as a task union. A reconfigurable task array is similar to a C function-pointer array. It maintains a list of task identifiers. Unlike a C function-pointer array, it does not require an identical function signature among the instances in an array. This can be done because the array only maintains a list of the task identifiers, and they are verified against the corresponding task declarations during the construction of the array. In addition, task identifiers are not allowed to be overloaded as in C++, so the function arguments are not necessary. Therefore, a task array can have instances with different argument lists. To use the array identifier as an alias of the task identifier, the array index must be a compile-time constant in order to verify the task signature. Otherwise, if a variable array index has to be used, all the tasks within the same reconfigurable task array must have an identical signature.

Reconfigurable task struct. This construct, see Figure 3(b), maintains a set of task identifiers, and allows several tasks within the same construct to be simultaneously active on the hardware. This is to say a logical static reconfiguration using multiplexor switching is performed. A physical dynamic reconfiguration is carried out when more space is required by displacing out an instance of task on the hardware.

This is analogous to the memory paging mechanism in operating systems. Bookkeeping is required to keep track of which tasks are on the hardware. A parameter is defined to indicate the maximum number of tasks in a construct that can be allowed simultaneously on the hardware. Different strategies can be used to select which task is to be replaced. For example, first-in first-out or least frequently used. In addition, different strategies can also be used to decide when a task is loaded. For example, pre-load as many tasks as possible, or load when first use. More sophisticated strategies, such as statistical profiling or speculative pre-fetching, could also be used. It is however, due to the reconfiguration overhead, complicated strategies may not often be justified.

Situations that may favour the use of this construct occur when the size of the tasks are in similar order and there is extra space to accommodate an additional task on hardware. This is because in such cases, the scheduling and space requirement of the tasks can be more easily calculated at compile time.

Reconfigurable task union. This construct, see Figure 3(c), explicitly specifies that all the tasks described in the construct are mutually exclusive. This indicates that a physical dynamic reconfiguration has to be carried out.

Situations that may favour the use of this construct occur when the size of the tasks in a construct are varied significantly. The use of this construct reduces the amount of work in keeping track of the resource usage. In addition, it produces simple task scheduling.

Parametrisable data types and operators. Run-time parametrisation is supported by the dynamic data type [9]. A run-time parameter is described using the `dynamic` data-type qualifier. Optimised designs can be achieved by specialising a hardware block with the corresponding run-time parameters. Figure 5 shows an example using the dynamic data type.

Overloaded operators dealing with operations on dynamic data types are also introduced. These operators include assignment operators, arithmetic operators, relational operators and bitwise operators.

```
TASK  taskid(int 8  k) {
        dynamic int 8  x=0, y=0;

        while ( 1 ) {
           ...  // some processing involve x and y

           RTPCONF {
              x++;
              if (x > k) { y++;   x = 0; }
           }
        }
     }
```

Figure 5: An example of dynamic data type. Both the parameters *x* and *y* can have their values changed at run-time. An instantiation and reconfiguration of the task block using the updated value of the parameters will be initiated at the end of the RTPCONF block scope.

A task block using the dynamic data type will only be instantiated when the values of the corresponding run-time parameters are specified. In addition, changes to the value of the run-time parameters will implicitly trigger a run-time reconfiguration of the corresponding tasks.

It is, however, sometimes desirable to have several run-time parameters updated at the same time. It would be inefficient, if run-time reconfiguration is required every time a run-time parameter is changed. To allow the value of several run-time parameters to be changed before a reconfiguration is initiated, the block specifier RTPCONF is used. An instantiation and run-time reconfiguration of the task using the updated parameters will only proceed at the end of the RTPCONF block scope.

A task using the dynamic data type will be translated to a corresponding parametrised low-level HDL hardware block. Annotation will be added to the HDL description to indicate the run-time requirements and methods of parameter update.

Constraints. Our high-level description supports use of constraint through annotation. The `with` specifier block is used to describe a list of constraints. It can be used to facilitate the low-level tools in guiding the design layout, timing requirement and other optimisation constraints. This specifier can be attached to a hardware block. For example, a task or a block can be enclosed by the { and } delimiters.

Placement information is usually an important factor that affects the performance of run-time reconfiguration. It provides crucial information that can guide the low-level tools in performing partial reconfiguration. It allows locating only those portions of a design that will require reconfiguration. A partial bitstream will be generated with those bits that need to be changed. Otherwise, the whole design would have to be rebuilt at run time.

Placement attributes can be described using the `rloc` placement specifier. Similar to other low-level HDL de-

```
TASK  taskid( ... ) {
         { /* a functional block */ } with {rloc("x.X.y.Y")}
      }
```

Figure 6: Example of placement attributes. The values *X* and *Y* denote placement control expressions. While `x.` and `y.` specify Cartesian coordinates in unit of functional blocks within a same scope.

```
typedef int 8  BYTE;
TASK  Dilation(BYTE src[ImgHeight][ImgWidth],
               BYTE& dst[ImgHeight][ImgWidth], BYTE mask[MSize][MSize]);
TASK  Erosion(BYTE src[ImgHeight][ImgWidth],
              BYTE& dst[ImgHeight][ImgWidth], BYTE mask[MSize][MSize]);

void main(void)
{
   RECONF sync  Operation[] = {Dilation, Erosion};
   BYTE         img[ImgHeight][ImgWidth], tmp[ImgHeight][ImgWidth];

   while ( 1 ) {
      capture(img);
      Dilation(img, tmp, maskD);
      Erosion(tmp, img, maskE);
      output(img);
   }
}
```

Figure 7: An example `main` function. This example declares two tasks: *Dilation* and *Erosion*, which are some morphological filtering operations.

scriptions, placement attributes are specified for respective hardware blocks. Attributes can be specified as Cartesian coordinates, a method that is commonly used in various families of FPGA. Attributes specified by `x.` and `y.` denote the horizontal and vertical coordinates in task or functional block unit within the same scope. Example of placement attributes can be found in Figure 6.

All other forms of annotation using the `with` specifier will be translated to the HDL without interpretation.

Initial bitstream and start of processing. An initial bitstream contains the representation of all static and default tasks. The `main` function is used as in standard C. It denotes the start of processing and provides a means of scheduling tasks or specifying top-level control. The `main` function also implicitly denotes loading of the initial bitstream before the processing is actually started. As long as a bitstream is loaded, processing can be expected to have started. However, if it is desirable for the hardware to wait for a software host, a `sync` functional class specifier can be used before the `main` keyword. This specifier will initiate a blocking communication and requests a synchronization signal before the first reference of a hardware task.

Figure 7 shows an example `main` function. In this example, the arguments *src* and *mask* in the tasks provide input data, and the argument *dst* is the data output. These two tasks are assigned to the same reconfigurable construct *Operation*. Since *Dilation* is the first element in the construct, it is therefore a default task that forms part of the initial bitstream. As soon as the processing has entered into the `main` block scope, the initial bitstream is expected to be loaded. Since *Operation* has declared `sync`, any reference to the two tasks will require a synchronization signal right before and after the tasks. *Dilation* will start only after *capture* is finished. *Erosion* will start only after *Dilation* is finished. Similarly, *output* can start as soon as *Erosion* has finished. Indeed, since the two tasks are in the same reconfigurable construct, either one of the two tasks will be on the hardware at any time. Therefore, *Erosion* would not be started before *Dilation* has finished, even without using the `sync` specifier.

The top-level control, task scheduling, and synchronization information will be separated from the hardware task descriptions and then translated to the host control.

Interfacing run-time parametrisable cores. To allow reuse of existing cores, an interface that connects the external core to a design can be constructed. There are four types of data to be specified: input data paths, output data paths, compile-time parameters, and run-time parameters.

An interface is declared using the `interface` keyword. It is followed by an argument list, which specifies the four types of data that are mentioned earlier. Each interface has its own namescope, which is implicitly declared using the identifier of itself. To name an argument declared in an interface, use the identifier of the interface followed by a full-stop symbol and then the identifier of the argument. An example interface can be found in Figure 11.

5 Compilation

We illustrate the generality of our design methodology using two different compilation paths, based on two different design languages. The first path involves the Handel-C language which does not have reconfiguration support. The second path involves RTPebble [3], which allows compiling run-time parametrisable designs.

Our target platform is Xilinx's Virtex series FPGAs with the JBits API. This reconfiguration platform allows dynamic circuit modification through manipulation of bitstreams. It also supports partial reconfiguration and constructions of parametrisable core.

Designs are run on the Celoxica's RC1000-PP reconfiguration board, which is supported by the JBits' XHWF interface. The board also comes with a run-time library and a set of pre-defined C function interfaces.

Path 1. We use the Handel-C language, which supports a high-level programming style, to compile designs into synchronous hardware. Handel-C supports most of the standard C syntax, but does not support run-time reconfiguration. Figure 8 shows a design flow of the method.

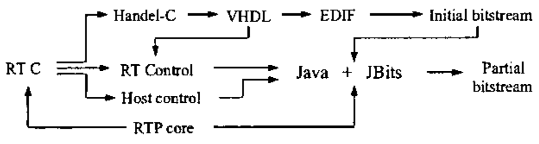

Figure 8: Compilation flow involving Handel-C.

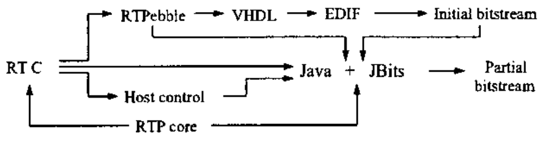

Figure 9: Compilation flow involving RTPebble.

We implement a parser that takes a design described in our high-level language and extracts the associated information. The information about the hardware design is described as hardware tasks, reconfiguration requirement is specified by the reconfigurable tasks, and run-time and software host control forms the skeleton that are not compiled to hardware. The parser generates the hardware part of a design in Handel-C, and the run-time and host control in a description file. The hardware design is compiled to VHDL using the Handel-C compiler. Since Handel-C does not support placement constraint, all the placement information is lost. The VHDL output is then required to piece together with the run-time control information to produce a Java program for the JBits platform. This is a manual process, in which each VHDL block is converted to a corresponding description using JBits, and has little control in the generation of hardware blocks in VHDL. Partial automation can be achieved by converting VHDL to Pebble. The process of re-inserting the placement attributes back to the VHDL output, however, can be automated by matching the identifiers of the generated hardware blocks.

Path 2. We use a low-level HDL that supports run-time parametrisation and generates Java program that uses the JBits API. This low-level HDL, RTPebble [3], can also produce output in VHDL in order to facilitate the production of an initial bitstream. Figure 9 shows a design flow of the method.

We use the same parser that is mentioned in *Path 1* to parse a design description. The generated Handel-C output is then hand coded to RTPebble with placement and run-time attributes intact. The run-time and host control information are then used to produce a Java host program that supplies the run-time parameters. The Handel-C to RTPebble translation process can be automated by a method such as [8]. This process will form part of our future work.

6 Case studies

This section evaluates our approach for unifying the instantiation and control of run-time parametrisable cores using a high-level language. We illustrate the capability of our approach through a video searching operation and a network firewall example.

```
TASK  SATM(int img[ImgHeight][ImgWidth], int tmpl[TmplHeight][TmplWidth],
           int mask[TmplHeight][TmplWidth], int& match) {

        for (int y=0; y < ImgHeight; y++) {
           for (int x=0; x < ImgWidth; x++) {

              int sum = 0;
              for (int i=0; i < TmplHeight; i++)
                 for (int j=0; j < TmplWidth; j++)
                    if (mask[i][j] == 1)
                       // accumulate difference over template:
                       sum += abs( img[i + y][j + x] - temp[i][j] );

              match[y][x] = sum;
           }
        }
     }
```

Figure 10: The SA-TM algorithm.

Video searching. We compare the implementations of a video searching method using reconfigurable designs constructed with our techniques against static-configuration designs synthesised with standard tools. This case study is intended to show: (1) how our techniques provide an effective framework in which to use run-time parametrised cores, and (2) how these cores can be used to improve the performance and area of designs described with high-level languages.

The Shape-Adaptive Template Matching (SA-TM) method allows retrieval of arbitrarily shaped objects from video streams [4]. This method measures the similarity of a template image at every position in every frame of a video stream. The similarity is measured by calculating the sum of absolute differences (SAD) on the luminance value of the pixels. Possible matches are identified when a position gives the minimum SAD value or when the SAD value is below a threshold. Both the search image and the template image are rectangular. A mask image can be used to specify an object in arbitrary shape. Only pixels that are masked are used for calculating the SAD. Figure 10 shows the SA-TM algorithm.

For a static-configuration implementation, in order to achieve acceptable performance-area trade offs over a range of object sizes, several factors have to be considered. For example, designer has to determine the size of arrays that are used to buffer the images, and the amount of resources to be shared.

A run-time reconfiguration design may involve partitioning the matching processes into several tasks. For ex-

```
interface  satm_rtp(int& 8 match, int 8 image, int 1 c=clk,
                    dynamic int tmpl[][], dynamic int mask[][]);

while( 1 ) {
   ...  // wait i/o or fetch template, mask and image data

   RTPCONF {
      satm_rtp.tmpl[0][0] = ... ;     satm_rtp.mask[0][0] = ... ;
      satm_rtp.tmpl[0][1] = ... ;     satm_rtp.mask[0][1] = ... ;
          ...                             ...
   } // reconfigure here using the updated run-time parameters

   while( notEndOfVideo ) {
      while( notEndOfFrame ) { // calc addr and fetch/store data
         sa_tm.image = image[y][x];
         match[y][x] = sa_tm.match;
         ...
      }
   }
   ...  // post-process results to determine best matches
}
```

Figure 11: A wrapper function that uses a SA-TM core. The interface *satm_rtp* declares the connection, while both *tmpl* and *mask* are run-time parametrisable parameters.

ample, matching of a search image can be divided into matching of the top half and the bottom half of an image. The reconfigurable constructs, described in Section 4, can be used to perform the reconfiguration. The two matching processes, top and bottom, will be performed in an alternating manner.

Given a static SA-TM core, a control mechanism is required to provide the search image from video input, to allocate storage from matching results, and to supply template and mask data for the matching processes.

In the case of a run-time parametrisable SA-TM core, additional control mechanisms are required. These mechanisms include supplying run-time parameters such as the template and mask images for dynamically specialising a circuit, initiating the run-time generation of a core and the management of hardware configuration processes. Other possible run-time parameters include the width and height of the array, and the horizontal and vertical position of a design. Figure 11 shows an example control code of the SA-TM core.

We have implemented two static-configuration SA-TM designs. They are both derived directly from the algorithm as shown in Figure 10 and are then parallelised by hand. Both the designs have their inner loops unrolled and memory accesses are parallelised by using image scan-line buffers. One of the implementation uses distributed RAMs as the scan-line buffers, and have the two innermost loops unrolled to form a combinational chain of additions. The other implementation uses an array that gets synthesised into shift registers. This array is used to store the summation results across the image scan lines. In addition, the two innermost loops are unrolled to an indirect form where the image pixels are broadcasted to the inner loop summations and the accumulated results are registered.

We have also implemented two run-time reconfigurable SA-TM designs. Both designs are hand placed. In one design, it features an additional pipelining on the image signal. This is to reduce the delay caused by large fanout when broadcasting image data. Both designs are specialised by supplying the template and mask data as run-time parameters. These parameters are used to configure the look-up tables and the routing inside configurable logic blocks (CLBs). The specialised designs reduce the amount of logic required to store and to process the templates and the masks. Figure 12 shows a floorplan of a run-time parametrisable design of the SA-TM. Pixels of the search image are streamed into the design horizontally one scan line at a time. These signals are then broadcasted to all of the processing elements.

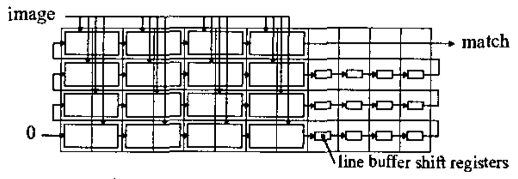

Figure 12: Floorplan of a run-time parametrisable design of SA-TM.

Table 1: Speed and area trade-offs for SA-TM designs.

		Speed / MHz	Area / slice	RTR time / ms
Static	distributed RAM	25	4049	-
	shift register	44	3573	-
RTP	shift register	38	1539	26
	pipelined	59	2115	26

All our designs are implemented on a Xilinx Virtex FPGA (XCV1000-6). The reconfiguration time is calculated for a XCV1000 in SelectMap mode at 60MHz using partial reconfiguration. The software is executed on a Pentium 4 at 1.6GHz with 512MB RAM.

Table 1 shows the results of speed, area and reconfiguration overhead of the four SA-TM designs. The templates that used in the tests are 12×12 pixels, while the search images are 100×100 pixels.

The static-configuration design that uses distributed RAM is the most straightforward derivation from the original SA-TM algorithm. However, it provides the lowest speed and largest area. This design could be further optimised if the combinational additions could be pipelined and dedicated RAMs, such as the BlockRAMs in Virtex devices, could be used.

The run-time parametrisable design that uses shift registers provides less performance than the equivalent static-configuration design. However, the run-time parametrisable version uses less than half of the configurable resources in the static-configuration version. The majority of the delay in this run-time parametrisable design is due to

```
RECONF  firewalls[][]
   = {{Account_Propaganda, Engineering_Propaganda, Public_Propaganda},
     {Account_Siamese, Engineering_Siamese, Public_Siamese}};

if ( onCondition )  firewalls[department](performance);
```

Figure 13: An example of reconfiguring between different firewalls.

the large fanout caused by broadcasting the image signals. On the other hand, the run-time parametrisable pipelined design reduces this delay by pipelining the image signal. This has achieved an increase in speed and also a reduction in resource usage.

Since the Virtex device and our implementation of the run-time software in Java are not optimised for run-time parametrisation, a large reconfiguration overhead has resulted. However, this overhead can be offset by processing a large amount of data. This is the typical case for searching images in a video library such as the SA-TM operation.

Network firewall. A packet-filtering firewall processor performs packet matching based on filter rules. Different sets of filter rules may be used to protect different parts of an organisational network. A previous study [6] has shown that a firewall implementation that uses the Siamese Twins structure can be in some cases up to 75% more area efficient than the Propaganda structure. On the other hand, the Propaganda structure can achieve up to 230% of the speed of the Siamese Twins structure. Therefore, different performance and hardware optimisation issues may determine the implementation scheme as well as the set of filter rules to be loaded on to the hardware.

Figure 13 shows an example of how the reconfiguration construct can be used for switching between different firewall implementations. Various network management schemes combined with different performance trade-offs can also be involved in determining the reconfiguration.

7 Conclusion

We have presented a unified description language for specifying run-time reconfiguration and its associated run-time control mechanisms. Two compilation paths have been introduced for two different hardware design languages with the JBits API. We discuss case studies that use our high-level description language to describe run-time parametrisable template matching designs and a network firewall. The results show that our system can be used to achieve a trade-off between increased performance and reduction of resource usages.

Our current and future work includes automating the translation process from generated Handel-C output to RT-Pebble, extending the system to include additional data types such as floating point and a type checker, to incorporate self-reconfiguration [2], and to automate run-time allocation and placement [10].

Acknowledgements. The support of UK Engineering and Physical Sciences Research Council (Grant number GR/R 31409, GR/R 55931 and GR/N 66599), the Croucher Foundation, Celoxica Limited and Xilinx, Inc. is gratefully acknowledged.

References

[1] P. Bellows and B. Hutchings, "JHDL – An HDL for Reconfigurable Systems", in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 1998, pp. 175-184.

[2] B. Blodget, S. McMillan and P. Lysaght, "Lightweight approach for embedded reconfiguration of FPGAs", in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 399-400.

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

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

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

[6] T.K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu and N. Dulay, "Irregular Reconfigurable CAM Structures for Firewall Applications", in *Field Programmable Logic and Applications*, LNCS 2778, Springer, 2003, pp. 890-899.

[7] S. McMillan and C. Patterson, "JBits Implementations of the Advanced Encryption Standard (Rijndael)", in *Field Programmable Logic and Applications*, LNCS 2147, Springer, 2001, pp. 162-171.

[8] I. Page and W. Luk, "Compiling Occam into FPGAs", in *FPGAs*, W. Moore and W. Luk (eds), Abingdon EE&CS Books, 1991, pp. 271-283.

[9] S. Singh and P. James-Roxby, "Rapid Construction of Partial Configuration Datastreams from High-Level Constructs Using JBits", in *Field Programmable Logic and Applications*, LNCS 2147, Springer, 2001, pp. 346-356.

[10] G.B. Wigley, D.A. Kearney and D. Warren, "Introducing ReConfigME: An Operating System for Reconfigurable Computing", in *Field Programmable Logic and Applications*, LNCS 2438, Springer, 2002, pp. 687-697.

[11] 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*, IEEE Computer Society Press, 2003.