Abstract

A new codesign compiler, Dash, provides a co-synthesis and co-simulation environment for mixed FPGA/processor architectures. It compiles a C-like description to a solution containing both processors and custom hardware, and allows the descriptions of FPGA-based processors to be heavily parametrised. The user may add instructions to the processors, and the Dash software architecture allows the user to add facilities for targeting these extra instructions to the compiler. This system is being used to design a number of case studies, and a single-chip codesign of an Internet video game is used to illustrate the design flow.

1 Introduction

An opportunity exists at the moment to bring together three trends in the FPGA, design automation and adaptive computing communities.

The first of these trends is the growing interest in implementing processor cores on FPGAs. The current interest in marketing intellectual property is driving a new industry where FPGA-based cores are being developed, sold and used in very short turn-around times. Some currently available cores allow small additions to be made to a basic processor by hand, allowing limited application-specific customisation.

At the same time the design automation industry has been developing techniques for compiling system level descriptions into a mix of hardware and software. Tools which allow mixed hardware/software systems to be developed and co-simulated in a single framework are becoming available commercially. Tools to automatically partition system descriptions into hardware and software are the subjects of research, although opinions differ as to whether this is better done by an automatic tool or a skilled designer with tool support.

The third of these three trends is the interest in architectures which combine a processor and FPGA. The FPGAs in these architectures could be used for anything from custom interfaces and preprocessing of incoming data to hardware acceleration as a configurable co-processor.

The opportunities that exist to bring together these trends are as follows:

- There is a lack of tools to target mixed FPGA/processor architectures. System design is a difficulty when users of such architectures are skilled hardware designers, as is likely to be the case when designing custom on-chip interfaces on the FPGA; it becomes a huge problem if the aim is to use this architecture for general purpose computing, when the user is likely to be a programmer, with no hardware design skills. In this case part of the FPGA can be committed to accelerate part of a user’s program. This program may be legacy code, and the user will not be willing or able to rewrite it for a specialised hardware implementation.

- There are even fewer tools to target mixed hardware/software systems where the software runs on FPGA-based processors, and the hardware is implemented on the same FPGAs. There are enormous opportunities for design trade-offs here, as the processor can be parametrised to fit the application. The problem becomes not just one of hardware/software partitioning, but partitioning while changing the power, size and instruction set of the processor, or even the number of processors.

This paper describes a new system which is being developed to provide a software flow for a new generation of such FPGA-based designs. The Dash (Design Automation for Software and Hardware) compiler can generate both software and hardware, and implement hardware/software codesign systems on a single FPGA. This system can be used to target a system containing parametrisable FPGA-based processors together with custom hardware. It can also be used for predesigned processors which are connected to an external FPGA or have an FPGA embedded in the architecture.
The paper is organised as follows. Section 2 describes previous work in these areas. Section 3 describes the Dash compiler, and Section 4 describes components of one of the case studies being used to develop the Dash flow: an Internet video game.

2 Related work

2.1 Reconfigurable computing

The Dash compiler builds on the work carried out in Oxford's Hardware Compilation Group. This work started with Page and Luk's paper [17] which described how Handel, a language based on occam, was suitable for describing hardware at the register transfer level, and could be compiled to a gate level netlist using a simple syntax-directed approach. In [16] Page described how this language could also be compiled to FPGA-based processors, which could be parametrised. In [13] the use of occam and Handel to design hardware/software systems was discussed. Since then the Handel approach has been developed and tested on a number of case studies. It has now been developed into a C-like language, Handel-C, together with a simulator and FPGA compiler that can target FPGA netlists or structural-level VHDL [2].

In [8] Gokhale and Stone consider the problem of compiling for an architecture containing a fixed RISC processor and FPGA resource. Researchers at Berkeley are designing a hybrid MIPS architecture and Java-based support that contains a reconfigurable resource [12]. The Raw project is developing compiler techniques that target a tiled architecture where the components are hybrid FPGA/RISC processors [19].

Weinhardt [20] proposes a high-level language programming approach for reconfigurable computers. This automatically partitions the design between hardware and software, and synthesises pipelined circuits from parallel FOR loops.

A great variety of boards have been developed which enable FPGAs to be added to general purpose computers. A survey is available at [9]. The most popular boards currently contain FPGA resources and memory on a PCI card, which can be plugged into a PC or other computer with a PCI bus. PMC mezzanine cards can be used to add extra resources or I/O. A survey of the literature on application specific instruction set processors and custom computing machines is contained in [11].

2.2 Hardware/software codesign

The problem of hardware/software codesign has been addressed by several research groups.

Some target a general architecture containing standard processors and custom hardware, and leave the partitioning decisions to the user. Examples of such systems are Polis [4], Partif [14], and Castle [3].

Other systems, such as Cosyma [5], Vulcan [10], Spec-syn [7], Tosca [1], and Lycos [15], target a more constrained architecture, and provide automated partitioning.

This work all targets standard, fixed, processors, and the user cannot take advantage of the parametrisation and customisation possible with FPGA-based processors.

3 The compiler

3.1 Overview

The Dash flow is shown in Figure 1. The user writes a description of the system in a C-like language. Blocks of code are allocated by the user to hardware or to software running on a given processor. Parts of the description which have been allocated to hardware can be written at a register transfer level, by using a version of the input language with a well defined timing semantics, or the scheduling decisions can be left to the compiler. The behaviour of this top-level description can be verified using the Dash simulator.

Dash then schedules and allocates any behavioural parts of the hardware description, and compiles the software description to assembly code. It also writes a parametrised description of the processors to be used, which may also have been designed by the user. The scheduled hardware, register transfer level hardware, software and processor descriptions are then combined. This allows a cycle-accurate co-simulation to be carried out, using the Handel-C simulator, and estimates of the speed and area of each statement in the hardware description are given. The user can then re-partition and go through this design cycle again if required. Once the design estimates are satisfactory the user can compile, optimise and then place and route the design using the FPGA vendor’s tools, again iterating if the more detailed performance and area figures indicate that design constraints are not met.

The target architecture for this system is an FPGA containing one or more processors, and custom hardware. The processors may be of different architectures, and may communicate with each other and with the custom hardware.

The Dash compiler may be used for the following:

1. To make use of parametrisation and instruction addition and removal for optimal core design. Dash provides an environment in which an FPGA-based processor and its compiler can be developed in a single framework.

2. To generate designs containing multiple communicating cores. The Dash compiler can target a heterogeneous set of communicating parametrised custom pro-
3. When hardware is required to run in parallel with the cores to meet speed constraints. Time critical parts of the system can be allocated to custom hardware, which can be designed at the behavioural or register transfer level.

4. When a compact logic block is required to perform non-time-critical tasks. Non-time critical parts of the design can be allocated to software, and run on a small, slow processor.

5. For auto-generation of circuitry on dynamic FPGAs. The FPGA can contain a small processor which can configure and reconfigure the rest of the FPGA at run time.

6. To explore efficient system implementations, by allowing parametrised application-specific processors with user-defined instructions to communicate with custom hardware. This combination of custom processor and custom hardware allows a very large design space to be explored by the user.

The remainder of this section describes the compiler and the compilation process in more detail.

### 3.2 The language

The input language to Dash is C-like, with some additions which allow efficient translation to hardware and parallel processes. These additions include the following:

- Variables are declared with explicit bit widths. This allows them to be efficiently implemented in hardware. The width of the processor may be declared, or else is calculated as the width of the widest variable which it uses. Operators are provided to manipulate bit fields in expressions. These operators can concatenate variables together, and take or drop bit fields within the expression.

- The `par` statement has been added to describe process-level parallelism. The Dash behavioural compiler can automatically extract fine-grained parallelism from the program, but generating coarse-grained parallelism automatically is far more difficult. Consequently Dash allows the user to express parallelism in the input language. Parallel processes can communicate using blocking channel communication. The keyword `chan` declares these channels.

- Attributes can be added specifying whether a block is to be put in hardware or software. For hardware the
attribute also specifies whether the description is to be interpreted as an RT- or behavioural-level description. For software the attribute also specifies the target processor. Other researchers have used pragmas to specify the hardware/software partitioning; attributes are preferred because it is impossible to generate #pragma from a macro, and because #pragma might be interpreted differently by other compilers [6].

3.3 Compiler front end

The compiler is designed in an object oriented way, and actually provides a class hierarchy of compilers, as shown in Figure 2. Each node in the tree shows a class which is a subclass of its parent node.

The top-level compiler class provides methods common to both the hardware and software flows, such as type-checking, and a system-level simulator. These methods are inherited by the hardware and software compilers, and may be used or overridden. The compiler class also specifies other, virtual, functions which must be supplied by its subclasses. So, the compile method on the hardware compiler class compiles a description to hardware; the compile method on the Processor A compiler compiles a description to assembly language which can run on Processor A.

The first step in the Dash flow is to parse and type check input code, perform some syntax level optimizations, and then a specific compiler can be attached to each block of code. There are two ways of specifying this:

- In command line mode. The compiler is called from the command line in the normal way. The program contains attributes specifying which compiler to use for a block of code.
- Interactively. A shell-like environment is provided, where the user has access to a set of functions within Dash. The user can call functions to estimate speed and size of hardware and software implementations, manually attach a compiler to a block of code, and call the simulator. This shell-like environment also allows complex scripts, functions and macros to be written and saved by the user.

An experimental version of the compiler is being developed which can automatically partition the program and generate the necessary interfaces.

3.4 Hardware compilation

The next stage is software or hardware specific. The parts of the description to be compiled into hardware use Dash-h, which is a behavioural synthesis compiler. The description is translated to a control/data flow graph, scheduled and bound, optimised, and then an RT-level description is written.

Many designers want to have more control over the timing characteristics of their hardware implementation. Consequently the Dash flow also allows the designer to write descriptions of the hardware at the register transfer level, and so define the cycle-by-cycle behaviour.

Dash allows users to describe such designs using C, by adding the capability of using a C-like description with a well defined timing semantics, Handel-C. In this description each assignment takes one clock cycle to execute, control structures add only combinational delay, and communications take one clock cycle as soon as both processes are ready. An extra statement is added to this RT-level version of the language: delay is a statement which uses one clock cycle but has no other effect. par may be used to specify statements which should be executed in parallel. Writing the description at this level, together with the ability to define constraints for the longest combinational path in the circuit, gives the designer close control of the timing characteristics of the circuit when this is necessary. It allows, for example, closer reasoning about the correctness of programs where parallel processes write to the same variable. This extra control has a price: the program must be refined from the more general C description, and the programmer is responsible for thinking about what the program is doing on a cycle-by-cycle basis. An example of a description at this level is discussed in Section 3.5.

3.5 Processor design

The description of the processor may itself be written in the C-like input language and compiled using Dash. As it is such an important element of the final design most users will want to write it at the register transfer level, in order to hand-craft important parts of the design. Alternatively the user may use predefined processors, or write the description
in VHDL or even at gate level, and merge it into the design using downstream tools.

With this flow the user can parametrise the processor design in nearly any way that he or she wishes. Of course it is important for the software compiler to be able to target the processor, and this is discussed in the following section.

The first parametrisation to consider is removing redundant logic. Unused instructions can be removed, along with unused resources, such as the floating point unit or expression stack.

The second parametrisation is to add resources. Extra RAMs and ROMs can be added. The instruction set can be extended from user assigned instruction definitions. Power-on bootstrap facilities can be added.

The third parametrisation is to tune the size of the used resources. The bit widths of the program counter, stack pointer, general registers and the opcode and operand portions of the instruction register can be set. The size of internal memory and of the stack or stacks can be set, the number and priorities of interrupts can be defined, and channels needed to communicate with external resources can be added. This freedom to add communication channels is a great benefit of codesign using a parametrisable processor, as the bandwidth between hardware and software can be changed to suit the application and hardware/software partitioning.

Finally, the assignment of opcodes can be made, and instruction decoding rearranged.

The user may think of other parametrisations, and Dash is designed so that it should be flexible enough to support many more possibilities.

The description of a very simple stack-based processor in this style is shown in Figure 3. This processor is in fact part of a direct implementation of the abstract syntax of the software compiler. It is described here not because it is a very interesting processor in itself, but to illustrate how processors can be parametrised, and to show an RT-level description.

The processor starts with a definition of the instruction width, and the width of the internal memory and stack addresses. This is followed by an assignment of the processor opcodes. The next line defines the program ROM, where the preprocessor includes `prog.o`, which is the output of the Dash software compiler. Next the registers are defined; the declaration `unsigned x y, z` declares unsigned integers `y` and `z` of width `x`. The program counter, instruction register and top-of-stack are the instruction width; the stack pointer is the width of the stack.

After these declarations the processor is defined. This is a simple non-pipelined two-cycle processor. On the first cycle (the first three-line par), the next instruction is fetched from memory, the program counter is incremented, and the top of the stack is saved. On the second cycle the instruction is decoded and executed. In this simple example a big `switch` statement selects the fragment of code which is to be executed.

This naive example illustrates a number of points. Various parameters, such as the width of registers and the depth of the stack can be set. Instructions can be added by including extra cases in the switch statement. Unused instructions and resources can be deleted, and opcodes can be assigned.

The example also introduces a few features of the register transfer level language which have not yet been mentioned, such as `rom` and `ram` declarations.

### Software compilation

Software parts of the description are compiled using Dash-s, a standard software compiler which has been organised in an object oriented way to allow users to add support for different processors (see Figure 2) and for processor parametrisations. For example, unused instructions from the processor description can be automatically removed, and support for additional instructions can be added.

The standard version of Dash-s supports many processor parametrisations. More complex and unexpected modifications are supported using the object oriented design of the compiler, which allows small additions to the back end to be made quickly by the user. Most of the mapping functions can be inherited from existing processor objects, and then minor additions can be made. The shell-like version of Dash allows a user to add these modifications using documented module interfaces without access to the implementation of the internal data structures. This has two advantages: it could be used to protect the intellectual property of a commercial version of the Dash compiler, and the interfaces document the ways the user should consider adding back-end improvements for the new processor.

### Co-simulation and estimation

The hardware and software is now brought back together into a single register transfer level description.

The following parts of the flow are currently carried out using the Oxford Handel-C system, although there is no reason that a standard VHDL or Verilog simulator and compiler could not be used.

Handel-C provides estimation of the speed and area of the design, which is written as an HTML file to be viewed using a standard browser, such as Netscape. The file shows two versions of the program: in one each statement is coloured according to how much area it occupies, and in the other according to how much combinational delay it generates. The brighter the colour for each statement, the greater the area or delay. This provides a quick visual feedback to the user of the consequences of design decisions.
void sw()
{
#define iw = 12; /* instruction width */
#define mw = 3; /* memory width */
#define CONST = 0 /* push constant */
#define LOAD = 1 /* push variable */
#define GLOBAL = 2 /* push address */
#define PUTCHAR = 15 /* put a character along the standard output channel*/
#define GETCHAR = 16 /* get a character from the standard input channel */

...}

Figure 3. Register transfer level description of simple processor

The Handel-C simulator is a fast cycle-accurate simulator which uses the C-like nature of the specification to produce an executable which simulates the design. It has an X-windows interface which allows the user to view VGA video output at about one frame per second.

When the user is happy with the RT-level simulation and the design estimates then the design can be compiled to a netlist. This is then mapped, placed and routed using the FPGA vendor’s tools.

3.8 Implementation language

Dash was written in objective CAML [18]. Objective CAML is a strongly typed functional programming language which is a version of ML. Importantly, Objective CAML provides a full range of imperative features, such as updatable arrays and imperative variables, which can be used where a pure functional approach would be inefficient. It fully supports the object oriented paradigm, and the module system provides a fine tuning over data encapsulation.

Objective CAML is a safe language, and the compiler performs many checks before compilation, which can eliminate many programming errors. It contains automatic memory management and incremental garbage collection, and provides tools for generating lexers and parsers.

This combination of features has allowed the rapid development and refinement of the Dash codesign compiler, with fewer errors than if it had been written in a more widely used language such as C.

3.9 Provable correctness

A subset of Dash could be used to provide a provably correct compilation strategy. This subset would include the channel communication and parallelism of occam and CSP. A formal semantics of the language could be used together with a set of transformations and a mathematician, to develop a provably correct partitioning and compilation route. There does not currently seem to be a strong demand for these systems, particularly for FPGA-based implementations, but this possibility of developing a provably correct codesign strategy exists and may have a larger role in the future.

4 Case study: an internet video game

A number of applications are being designed and built using the tools and methodology described above. These applications include software acceleration of a logic minimization program, video processing, and computer networking.

To illustrate the methodology using an application which is small enough to describe here a simple Internet video
game was designed. The user of this video game can fly a plane over a detailed background picture. Another user can be dialled up, and the screen shows both the local plane and a plane controlled remotely by the other user. This is not an exceptionally difficult problem, but the main challenge is that the system must be implemented on a single medium-sized FPGA.

4.1 Implementation platform

The platform for this application was a generic and simple FPGA-based board which has been designed and built at Oxford. A block diagram of the board is shown in Figure 4, and a picture is shown in Figure 5.

The Hammond board contains a Xilinx 4000 series FPGA and 256kb synchronous static RAM. Three buttons provide a simple input device; alternatively a standard computer keyboard can be plugged into the board. There is a parallel port which is used to configure the FPGA, and a serial port. The board can be clocked at 20 MHz from a crystal, or from a PLL controlled by the FPGA. Three groups of four pins of the FPGA are connected to a resistor network which gives a simple digital to analogue converter, which can be used to provide 12 bit VGA video by implementing a suitable sync generator on the FPGA.

4.2 Problem description and discussion

The specification of the video game system is as follows:

- The system must dial up an Internet service provider, and establish a connection with the remote game, which will be running on a workstation.
- The system must display a reconfigurable background picture.
- The system must display on a VGA monitor a picture of two planes: the local plane and the remote plane.
- The position of the local plane will be controlled by the buttons on the Hammond board.
- The position of the remote plane will be received over the dialup connection every time it changes.
- The position of the local plane will be sent over the dialup connection every time it changes.

This simple problem is interesting because it combines some hard timing constraints, such as sending a stream of video to the monitor, with some complex tasks without timing constraints, such as connecting to the Internet service provider. There is also an illustration of contention for a shared resource, which will be discussed later.

4.3 System design

A block diagram of the system is shown in Figure 6. The system design decisions were quite straightforward. A VGA monitor is plugged straight into the Hammond board. To avoid the need to make an electrical connection to the telephone network a modem was used, and plugged into the
serial port of the Hammond board. Otherwise it would have been quite feasible to build a simple modem in the FPGA. The subsystems required are

- serial port interface,
- dial up,
- establishing the network connection,
- sending the position of the local plane,
- receiving the position of the remote plane,
- displaying the background picture,
- displaying the planes.

A simple way of generating the video is to build a sync generator in the FPGA, and calculate and output each pixel of VGA video at the pixel rate. The background picture can be stored in a "picture RAM". The planes can be stored as a set of 8x8 characters in a "character generator ROM", and the contents of each of the characters positions on the screen stored in a "character location RAM". These design decisions were partly based on the reuse of prewritten library code to perform some of these functions.

4.4 Hardware/software partitioning

The hardware portions of the design are dictated by the needs of some part of the system to meet tight timing constraints. These are the video generation circuitry and port drivers. Consequently these were allocated to hardware, and the C descriptions written at register transfer level to enable them to meet the timing constraints. The picture RAM, character generator ROM and character location RAM were all stored in the Hammond board RAM bank, as the size estimators showed that there would be insufficient space on the FPGA.

The parts of the design to be implemented in software are the dial up negotiation, establishing the network, and communicating the plane locations. These are non-time critical, and so can be mapped to software. The program is stored in the RAM bank, as there is not space for the application code in the FPGA.

The main function is shown in Figure 7. The first two lines declare some communication channels. Then the driver for the parallel port and sync generator are started, and the RAM is initialised with the background picture, the character memory and the program memory. The parallel communicating hardware and software process are then started, communicating over a channel hwswchan. The software establishes the network connection, and then enters a loop which transmits and receives the position of the local and remote plane, and sends new positions to the display process.

```c
void main()
{
    chan hwswchan;
    chan unsigned 8 port;

    par {
        parallel_port(port);
        SyncGen();
        {
            initialiseRam(port);
            par {
                display(hwswchan);
                sw(hwswchan);
            }
        }
    }
}
```

Figure 7. RTL description of main

4.5 Processor design

The simple stack-based processor from Section 3.5 was parametrised in the following ways to run this software. The width of the processor was made to be 10 bits, which is sufficient to address a character on the screen in a single word. No interrupts were required, so these were removed, as were a number of unused instructions, and the internal memory.

4.6 Co-simulation

The RT-level design was simulated using the Handel-C simulator. Sample input files mimicking the expected inputs from the peripherals were prepared, and these were fed into the simulator. A black and white picture of the colour display is shown in Figure 8 (taken as a snapshot of the X-window drawn by the co-simulator).

The design was then placed and routed using the proprietary Xilinx tools, and successfully fit into the Xilinx 4013 FPGA on the Hammond board. This application would not have been possible to implement without the codesign system. A hardware-only solution would not have fitted onto the FPGA; a software-only solution would not have been able to generate the video and interface with the ports at the required speed.

4.7 Real world complications

The Dash flow was presented with an implementation challenge with this design which merits some discussion. The processor had to access the RAM (because that is where the program was stored), whilst the hardware display process simultaneously had to access the RAM because this is where the background picture, character map and screen
Figure 8. Simulation display

map were stored. This memory contention problem was made more difficult to overcome because of an implementation decision made during the design of the Hammond board: for a read cycle the synchronous static RAM which was used requires the address to be presented the cycle before the data is returned.

The display process needs to be able to access the memory without delay, because of the tight timing constraints placed on it. A semaphore is used to indicate when the display process requires the memory. In this case the processor stalls until the semaphore is lowered. On the next cycle the processor then presents to the memory the address of the next instruction, which in some cases may already have been presented once.

The designer was able to overcome this problem using Dash because of the facility for manual partitioning and describing parts of the design at the register transfer level.

It is hoped that in the future a sophisticated system with automated partitioning will be able to overcome this problem, or that it could be solved using timing constraints placed on the behavioural description. However, in the meantime using our approach the designer has the control necessary to enforce his or her view of the best implementation on the compiler without leaving the Dash framework, and this has been achieved with minimal complications to the flow so as not to put off the novice user.

5 Conclusion

The Dash compiler provides co-simulation and co-synthesis support for implementing communicating hardware and parametrised processors on FPGAs. It is being used to rapidly explore trade-offs in some practical application areas, such as video processing, networking, and hardware acceleration. The unique feature of the system is that single-chip hardware/software systems can be quickly developed and implemented in a single framework.

The philosophy behind Dash is to give the designer the freedom to describe the system required. So it allows designer to write a behavioural description of the system in C, to specify hardware and software partitions, and to describe parts of the hardware partition in a C-like language at the register transfer level.

An open object oriented structure and a shell-like interactive version of the system allows the user to experiment with partitions and to add compiler support for extra instructions which may be added to the processor as part of the parametrisation process.

There is much scope for future work in this area. An automated partitioning system is under development. The aim is to provide an automated system which can take advantage of processor parametrisation with custom instructions, variable width registers, and multiple execution units, as well as assigning operations to hardware or software. FPGA-based codesigns allow a very large design space to be explored, and the opportunity to provide a very high communication bandwidth between the processors and the hardware will mean that codesign solutions have a good chance of producing more efficient designs.

The implementation of more case studies is continuing to drive new developments of the compiler.

Acknowledgements

All of this work was carried out with the support of my colleagues in the Hardware Compilation Group at Oxford. Matt Aubury was an invaluable source of ideas and discussion, and was the developer of the Handel-C compiler and simulator. Matt Bowen helped with implementation problems, and wrote the serial interface for the Hammond Board. The detailed design of the Hammond Board itself was carried out by Dominic Plunkett. Matthias Sauer provided many ideas during the initial development of Dash. Jonathan Saul is supported by the EPSRC as an Advanced Research Fellow, under grant B/96/AF/2176.

References


