Towards a Declarative Framework for Hardware-Software Codesign

Wayne Luk and Teddy Wu
Programming Research Group
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, England OX1 3QD

Abstract

We present an experimental framework for mapping declarative programs, written in a language known as Ruby, into various combinations of hardware and software. Strategies for parametrised partitioning into hardware and software can be captured concisely in this framework, and their validity can be checked using algebraic reasoning. The method has been used to guide the development of prototype compilers capable of producing, from a Ruby expression, a variety of implementations involving field-programmable gate arrays (FPGAs) and microprocessors. The viability of this approach is illustrated using a number of examples for two reconfigurable systems, one containing an array of Algotronix devices and a PC host, and the other containing a transputer and a Xilinx device.

1 Introduction

Although it has been known for many years that, from a functional point of view, there is little distinction between hardware and software, in current practice they are mostly developed using very different methods and tools. This paper presents a coherent framework for describing and producing implementations that contain one or more hardware and software components. Our aim is to investigate the features for a system-level language to provide a rapid, reliable and cost-effective route for realising such designs. The purpose of this paper is to provide an overview of our techniques and tools, many of which are still under development.

Much of our work is based on a declarative language known as Ruby (see [4], [10]). Ruby has been used principally as a hardware description language in the past. Here we shall explore its use as a system design language. Our approach is particularly applicable to designs with a uniform structure, which can be found in, for instance, many signal processing systems.

An overview of the design process in our current framework is shown in Figure 1. The user first prepares a Ruby program for the desired computation, indicating which parts should be implemented in hardware and which parts should be implemented in software. Our system-level compiler produces from this program an implementation involving one or more hardware and software components, depending on the target system. The appropriate interface will be included in these components to allow them to communicate with each other. Other hardware and software tools can then be used to realise the implementation; for instance, vendor-provided implementation tools can be used to generate configuration data for field-programmable devices.

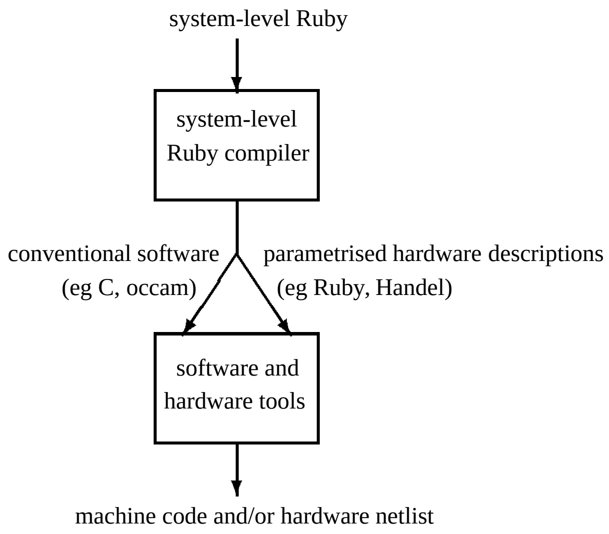

system-level Ruby
system-level Ruby compiler
conventional software
(eg C, occam)
parametrised hardware descriptions
(eg Ruby, Handel)
software and hardware tools
machine code and/or hardware netlist

Figure 1 An overview of the design process in our framework.

At present the partitioning into hardware and software is carried out by the user, and we are exploring possibilities for automating this process. While a fully-automatic design tool can sometimes be very useful, we are aware of the importance for the user to retain full control whenever the need arises. Hence our framework is evolved in such a way that it should be possible to use our techniques in conjunction with other methods, or to optimise generated designs further by hand or by other tools.

Let us explain in some detail the motivations behind the use of Ruby. Declarative languages often have a simple semantics that makes them easy to understand and to use [2]. Ruby shares this advantage, and it has additional features that are attractive for system design. First, there are a number of primitives and functions in Ruby for describing parametrised designs concisely; they allow the user to focus on the essential structure of the system and also serve as high-level design documentation. Second, as we shall see later, a Ruby expression can be implemented in a number of ways, ranging from a sequential program to a piece of fast parallel hardware – or a combination of both; this flexibility provides the basis for a unified representation of heterogeneous systems. Third, Ruby has an algebra for transforming designs based on simple equational reasoning, so one can generate from an initial description a variety of designs customised to specific requirements. This facility is particularly useful in checking the correctness of parametrised transformation strategies, such as those for partitioning a program into hardware and software parts. Finally, data refinement is supported in our framework, thus a designer may start with, for instance, a program involving integer datatypes and explores the effects of different bit-level representations, using formal techniques or simulation.

It should perhaps be made clear that our work is intended to complement, rather than to replace, existing hardware and software languages and tools. Our framework can be used in an incremental way to produce and to assess designs rapidly; once a promising implementation is found, then other tools can be used to further optimise it if desired.

2 Primitives and compositions

This section introduces the way that computation and wiring components are described in Ruby, and the composition functions which allow connection of components with a compatible interface.

A design is represented in Ruby by a binary relation of the form $x\ R\ y$ where x and y belong respectively to the domain and range of R. For instance, a squaring operation can be described by

$$x\ sq\ y \quad \Leftrightarrow \quad x^2 = y$$

or, more succinctly, by $x\ \ sq\ \ x^2$. In this paper we shall focus on designs with inputs in the domain and outputs in the range, so we can also define such components in the usual way as in $sq\ x\ =\ x^2$.

Wiring relations can be used to replicate, extract or rearrange data. As an example, the relation $fork$ can be used to duplicate a datum, since $x\ \ fork\ \ \langle x, x\rangle$. Extracting an element from a pair is achieved by the projection relations π_1 and π_2, defined by $\langle x, y\rangle\ \pi_1\ x$ and $\langle x, y\rangle\ \pi_2\ y$. Some wiring relations are parametrised: for example, zip_n relates a pair of sequences of the same length to a sequence of pairs,

$$\begin{aligned}&\langle\langle x_0, x_1, \ldots, x_{n-1}\rangle, \langle y_0, y_1, \ldots, y_{n-1}\rangle\rangle \\ &\qquad zip_n\ \ \langle\langle x_0, y_0\rangle, \langle x_1, y_1\rangle, \ldots, \langle x_{n-1}, y_{n-1}\rangle\rangle.\end{aligned}$$

Wiring relations not only shorten description of data rearrangement – they also have useful properties that can be used for optimising designs which will be explained later.

Two components Q and R can be joined together by sequential composition if they share a common interface s which is hidden in the composite program $Q\,;R$ (Figure 2a),

$$x\,(Q\,;R)\,y \quad \Leftrightarrow \quad \exists s.\,(x\ Q\ s)\ \wedge\ (s\ R\ y). \qquad (1)$$

Many readers would recognise that sequential composition corresponds to relational composition. It is simple to show from this definition that sequential composition is associative. Notice that x, y and s can be composite: for instance Q can be add, where $\langle x, y\rangle\ \ add\ \ (x+y)$.

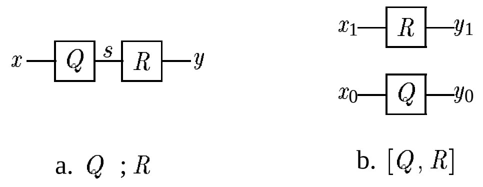


Figure 2 Sequential and parallel composition.

If there are no connections between Q and R, the composite is represented by parallel composition $[Q, R]$, where

$$\langle x_0, x_1\rangle\,[Q, R]\,\langle y_0, y_1\rangle \quad \Leftrightarrow \quad (x_0\ Q\ y_0)\ \wedge\ (x_1\ R\ y_1) \qquad (2)$$

as shown in Figure 2b. Given that ι is the identity relation ($x\ \iota\ x$), we use the abbreviations

$$\begin{aligned}\mathsf{fst}\ R &= [R, \iota], \\ \mathsf{snd}\ R &= [\iota, R].\end{aligned}$$

Repeated sequential composition of n copies of R is given by R^n, so $R^4\ =\ R\ ;\ R\ ;\ R\ ;\ R$. Similarly repeated parallel composition is given by $\mathsf{map}_n\ R$: $\mathsf{map}_3\ R\ =\ [R, R, R]$. There are functions for capturing other patterns of replicating components and for building state machines; details of these can be found in [4], [5], [10] and [12].

3 Design mapping

It is possible to implement a Ruby expression in a number of ways. For instance, the expression $Q\ ;\ R$ can be implemented in hardware as two connected circuit blocks as shown in Figure 2a, or implemented in software in the form of C and occam programs. In this case, if both Q and R are functions with inputs in their domain and outputs in their range, then the code for Q will be executed before the code for R. Repeated compositions can be translated into for-loops in software.

More interestingly, an implementation of $Q\,;R$ can have Q in hardware and R in software or vice versa, provided that the interface between them follows the sequencing constraint imposed by sequential composition. At present we adopt synchronous communication [3] between components executing in parallel – whichever completes execution first has to wait for its partner to finish. If there are several ways of interfacing between Q and R, the user should be able to indicate which one if the default option is not preferred; we are also exploring methods for automating the selection of alternative interfaces.

To map the components of an expression into hardware or software, we label them either by $\mathcal{H}$ or by $\mathcal{S}$, as in $\mathcal{H}\ Q\ ;\ \mathcal{S}\ R$. For convenience, $\mathcal{S}$ and $\mathcal{H}$ will be distributed through composite functions to their components and $\mathcal{S}\ (\mathcal{H}A) = \mathcal{H}A$ and $\mathcal{H}\ (\mathcal{S}B) = \mathcal{S}B$, so $\mathcal{S}\ (P\ ;\ [Q, \mathcal{H}\ R]\ ;\ S)$ is an abbreviation for $\mathcal{S}\ P\ ;\ [\mathcal{S}\ Q, \mathcal{H}\ R]\ ;\ \mathcal{S}\ S$. It is possible to have several physical hardware or software resources: the implementation for

$$\mathcal{S}\ (A\ ;\ fork\ ;\ [B, \mathcal{H}_0\ (fork\ ;\ [C, \mathcal{H}_1\ D])])$$

contains (a) a software program for implementing A, B and the first $fork$, specified by $\mathcal{S}$; (b) a circuit for implementing C and the second $fork$, specified by $\mathcal{H}_0$; and (c) another circuit for implementing D, specified by $\mathcal{H}_1$.

4 Design tools

We have developed a set of compilation tools for a subset of Ruby known as T. Currently a relation in T always has inputs in its domain and outputs in its range. Sequential primitives [10] such as $\mathcal{D}$ have not been implemented. Many of our tools are target-independent; the target-specific tools correspond to compiler backends that produce the output in a particular format.

The target-independent compiler converts a T program into two parts: a hardware part H, and a software part S. The execution of H and S can be simulated. Our simulator supports both symbolic simulation and numerical simulation involving integers and fixed-point numbers; a mixture of numerical and symbolic simulation is possible as well. The target-specific tools then generate device- or language-specific files from H and S. The hardware part H can be further processed by compilers [11] for various FPGA devices, including those from Xilinx and Algotronix. S, the software part, can be used to produce C or occam programs.

When designing with Ruby, one usually starts with a high-level description of the computation without deciding, for instance, how integers will be represented at bit-level. In a separate step, often known as data refinement, the implementation of high-level data structures is considered. This method is useful in structuring design documents, and in providing additional flexibility for realising designs. Our design tools contain functions that convert between data representations, facilitating the analysis of finite-precision effects. For instance, given that add is integer addition and $sint2bit_n$ and $bit2sint_n$ respectively convert integers to their n-bit two's complement representations and vice versa, the expression

$$[bit2sint_n, bit2sint_n]\ ;\ add\ ;\ sint2bit_n$$

models an n-bit adder. The trade-offs of adopting different data representation schemes, for both hardware and software, can be explored by systematically replacing high-level operations by their low-level models – a step which is being automated.

5 Partitioning strategies

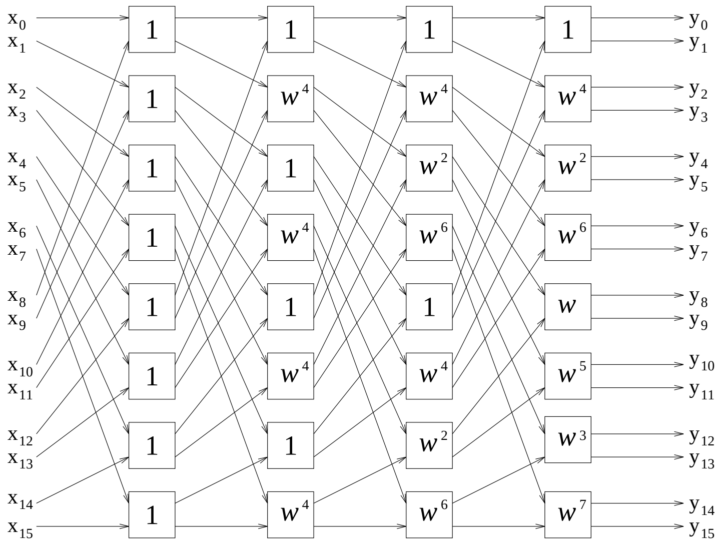
In accelerating performance-critical programs, a pure hardware implementation may not be attractive due to reasons such as cost; a mixed hardware-software implementation may be more appropriate. This section presents parametrised strategies capable of rearranging the computations to achieve an effective partitioning for hardware-software implementations. For simplicity we shall focus on the case that we have a single hardware unit and a single software unit, but it would be possible to develop partitioning strategies for multiple hardware and software units.

As our first example, consider implementing R^n on a piece of hardware that can only accommodate R^p, where $n > p$. One solution is to find q and r such that $n = p \times q + r$ in order to implement R^n as

$$\mathcal{S}\ ((\mathcal{H}\ (R^p))^q\ ;\ R^r). \tag{3}$$

There are two sequential loops in this implementation. The first contains q iterations in software, and each of these iterations invokes the hardware once for computing R^p; the second loop iterates r times to compute R^r in software.

Clearly the above solution assumes that the hardware implementation of R is faster than the software implementation, and that the time for communicating between the



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
1 1 1 1 1 1 1 1
1 w^4 1 w^4 1 w^4 1 w^4
1 w^4 w^2 w^6 1 w^4 w^2 w^6
1 w^4 w^2 w^6 w w^5 w^3 w^7
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12} y_{13} y_{14} y_{15}

Figure 3 Using a butterfly network to implement a 16-point fast Fourier transform.

hardware and the software components is much less than the computation time.

To describe a partitioning method for parallel composition, we need two more wiring operations. The first, $group_{m,n}$, relates a sequence of $m \times n$ elements to a sequence of m elements, each of which is a sequence of n elements, so $\langle 1,2,3,4,5,6\rangle\ group_{3,2}\ \langle\langle 1,2\rangle,\langle 3,4\rangle,\langle 5,6\rangle\rangle$. The second wiring operation, $app_{m,n}$, relates a sequence of two sequences of length m and n to the appended version, so $\langle\langle 1,2,3\rangle,\langle 4,5\rangle\rangle\ app_{3,2}\ \langle 1,2,3,4,5\rangle$. We shall also need the converse operation, given by

$$x\ R^{-1}\ y \quad\Leftrightarrow\quad y\ R\ x\,.$$

The pattern $P^{-1}\ ;\ Q\ ;\ P$ – in words 'Q conjugated by P' – will be abbreviated to $Q\backslash P$.

The theorem that equates a repeated parallel composition to its partitioned version is

$$\begin{aligned}&\mathsf{map}_{p\times(q+r)+s}\ R\\ &\quad= [(\mathsf{map}_p\ ([\mathsf{map}_q\ R,\ \mathsf{map}_r\ R]\backslash app_{q,r}))\\ &\qquad \backslash group_{p,q+r},\ \mathsf{map}_s\ R]\backslash app_{p\times(q+r),s}\,. \quad (4)\end{aligned}$$

While this theorem may appear complicated, it can be proved algebraically using simpler laws. We can now label the components on the right-hand side of this equation with $\mathcal{H}$ and $\mathcal{S}$, so that $\mathsf{map}_{p\times(q+r)+s}\ R$ can be implemented in hardware using only q copies of R:

$$\begin{aligned}&\mathcal{S}\ ([(\mathsf{map}_p\ ([\mathcal{H}\ (\mathsf{map}_q\ R),\ \mathsf{map}_r\ R]\backslash app_{q,r}))\\ &\qquad \backslash group_{p,q+r},\ \mathsf{map}_s\ R]\backslash app_{p\times(q+r),s}).\end{aligned}$$

There are two sequential loops in this implementation. One involves p iterations, each implementing concurrently $\mathsf{map}_q\ R$ in hardware and $\mathsf{map}_r\ R$ in software. The other loop involves implementing $\mathsf{map}_s\ R$ in software. To minimise idle time, the parameter r should be chosen such that the time for executing q copies of R in hardware is around the same as that for executing r copies of R in software. If necessary, serialisation strategies [10] can be applied to the hardware part to balance the hardware load and the software load.

Our next example involves partitioning of a butterfly network. We shall need riffle_n, a wiring operation similar to zip_n but without some of the internal sequence structures in the domain and range data, to describe a perfect shuffle architecture [5]:

$$\begin{aligned}&\langle x_0,x_1,\ldots,x_{n-1},y_0,y_1,\ldots,y_{n-1}\rangle\\ &\qquad \mathit{riffle}_n\ \langle x_0,y_0,x_1,y_1,\ldots,x_{n-1},y_{n-1}\rangle.\end{aligned}$$

One can show that $\mathit{riffle}_n = group_{2,n}\ ;\ zip_n\ ;\ group_{n,2}^{-1}$. A parametrised butterfly network can be defined using riffle_n:

$$\begin{aligned}\mathsf{bfy}_n\ R &= (\mathsf{bfycol}_n\ R)^{n+1},\\ \mathsf{bfycol}_n\ R &= \mathit{riffle}_{2^n}\ ;\ (\mathsf{map}_{2^n}\ R)\backslash group_{2^n,2}^{-1}.\end{aligned}$$

Figure 3 illustrates the use of a butterfly network to compute a 16-point fast Fourier transform. Each node in this network takes in a pair $\langle u,v\rangle$ and computes $\langle v-a\times u, v+a\times u\rangle$, where a is the coefficient for the multiplication (the twiddle factor) and is labelled at each node in the figure. This network can be described by an "indexed" version of bfy_3, ibfy_3, which takes into account the variation of coefficients at each node.

When a butterfly network is too large to fit onto a given piece of hardware, we can partition it in several ways. First, since each column in a butterfly network is the same, we can use Equation 3 to partition bfy. Second, if the amount of hardware is too small to implement even a single column of a butterfly network, we can use the theorem below to partition a bfycol:

$$\begin{aligned}\mathsf{bfycol}_{m+n}\,R \;&=\; \mathit{riffle}_{2^m} \backslash \mathit{group}^{-1}_{2^{m+1},2^n} \;;\\ &\quad (\mathsf{map}_{2^m}\;(\mathsf{bfycol}_n\;R))\backslash \mathit{group}^{-1}_{2^m,2^{n+1}}.\end{aligned}$$

An instance of this theorem is shown in Figure 4. The expression $\mathsf{map}_{2^m}\;(\mathsf{bfycol}_n\;R)$ on the right-hand side of the above equation can now be rewritten and optimised by Equation 4.

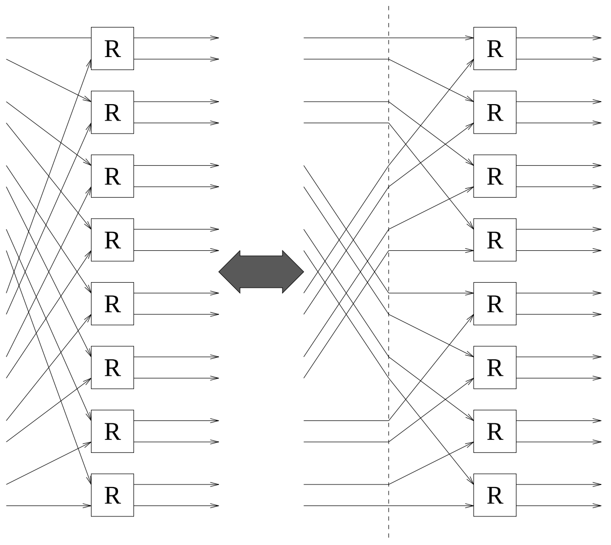


Figure 4 $\mathsf{bfycol}_{m+n}\,R \;=\; \mathit{riffle}_{2^m} \backslash \mathit{group}^{-1}_{2^{m+1},2^n}\;;$ $(\mathsf{map}_{2^m}\;(\mathsf{bfycol}_n\;R))\backslash \mathit{group}^{-1}_{2^m,2^{n+1}}$, with $m = 1$ and $n = 2$.

As a final example, we have expressed in Ruby a method for partitioning multidimensional programs [13]. The method involves a divide-and-conquer structure, with the "divide" and "merge" phases carried out by a general-purpose processor while the "conquer" phase is handled by application-specific hardware. It can be expressed in Ruby as:

$$\mathcal{S}\;(\mathit{divide}_n\;;\;\mathit{conquer}_n\;(\mathcal{H}\;R)\;;\;\mathit{merge}_n)$$

where R is an n-dimensional program. [13] presents the conditions such that the equation

$$\mathit{divide}_n\;;\;\mathit{conquer}_n\;R\;;\;\mathit{merge}_n \;=\; R \qquad (5)$$

is true.

The application of some of these partitioning strategies will be illustrated in the following sections. There are other partitioning methods, for instance those involving a combination of composition functions, which have not been described above.

6 Edge detection on the CHS2x4

The divide-and-conquer partitioning strategy has been used in implementing the Sobel edge detector on an FPGA-based hardware accelerator known as CHS2x4 [1] (Figure 5). The CHS2x4 is a full-length IBM AT card which communicates with the host computer through the AT bus. The board consists of three subsystems: the Computation Subsystem which holds eight CAL 1024 chips [6] arranged in a two by four array, the Memory Subsystem which contains 256 Kbytes of SRAM, and the Interface and Control Subsystem which deals with the communication between the board and its host machine. At present data transfer between the CAL chips and the on-board memory is restricted to sequential input and output over a byte-wide channel, controlled by invoking C-library routines provided by the manufacturer. Each CAL chip is an FPGA consisting of 32 by 32 cells. Each cell has a one-bit input port and a one-bit output port on each of its four sides. An input port can be programmed to connect to one or more output ports, or to a function unit which can be programmed to behave either as a two-input combinational logic gate or as a latch.

The Sobel edge detection algorithm involves a two-dimensional convolution using the masks

$$[[-1,-2,-1],[0,0,0],[1,2,1]]$$

and

$$[[-1,0,1],[-2,0,2],[-1,0,1]]$$

to produce the image gradient in the horizontal and in the vertical direction. The squares of the two gradients are then summed together and compared against a threshold. Our hardware implementation consists of adders, subtractors, registers, multiplexers, magnitude extractors and a multiplier. Partitioning is necessary because the size of many images requires a larger memory than is available in our CHS2x4 system.

Our implementation is guided by the Ruby expression

$$\mathcal{S}\;(\mathit{divide}_2\;;\;\mathit{conquer}_2\;(\mathcal{H}\;\mathit{Sed})\;;\;\mathit{merge}_2)$$

(from Equation 5). In this expression the component divide_2, the two-dimensional version of divide_n, divides an image into a series of smaller images, each of which is then processed by $\mathit{conquer}_2$ using the Hardware Sobel

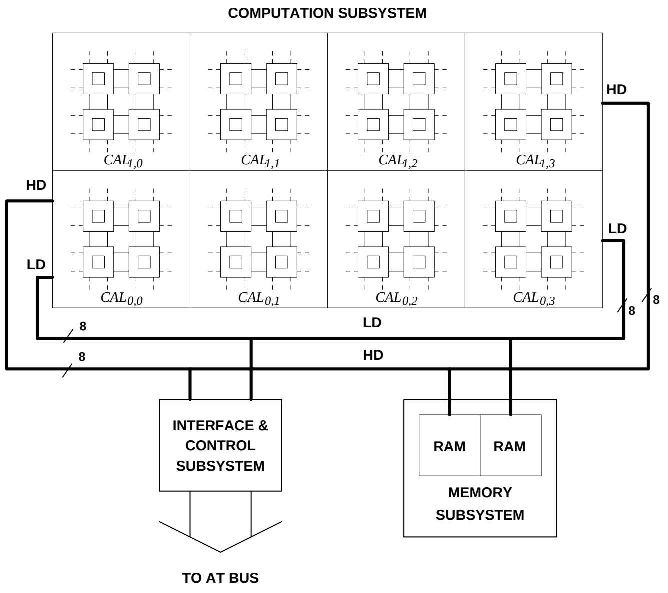


Figure 5 The CHS2x4 board.

edge detector Sed, and the resulting edge maps are stitched together in software according to $merge_2$. The result presented below has been obtained by an implementation produced by hand, but it should be straightforward to automate a substantial part of the work using our prototype compiler. In this case the hardware-software interface would be implemented by calls to the library routines supplied by Algotronix.

We have carried out some experiments to compare the Sobel edge detector in software on a 386-based PC against the FPGA-assisted version. Even with the slow software-controlled FPGA execution, a speedup of over 20 times can be obtained if data transfer overhead to and from disk is not included, while a speedup of a factor of two is observed if we include the data transfer overhead. In fact, the critical path of our hardware design is found to be around 574ns, which means that it should be capable of performing edge detection for images of 128 by 128 pixels at 35 frames per second, provided that data can be supplied at that rate. We have also found that the time for dividing the image into sub-images and combining the results is around 5% of the total processing time; so our partitioning strategy appears to incur a modest overhead for this system.

7 Implementing designs on HARP1

The HARP1 system is constructed as a platform for developing hybrid hardware/software systems [8]. It integrates a T805 transputer, 4Mbyte DRAM, a Xilinx 3090 (or 3195) FPGA chip with two local banks of 32K by 16-bit SRAM, and a 100MHz variable clock frequency synthesizer on an industry-standard (size 6) TRAM module (Figure 6). The speed of the FPGA depends on the critical path of the logic that it implements, and it can be varied using the frequency synthesizer. The board can be regarded as a prototype of a future microprocessor, which has a conventional RISC core closely integrated with a flexible hardware coprocessor.

The transputer can communicate with the FPGA in a number of ways. It can directly communicate with the FPGA through the bus, or it can use the SRAM or the DRAM as a shared memory with the FPGA. One common mode of operation of HARP1 is as follows. The transputer first loads the SRAM with data, and it then reconfigures the FPGA to carry out the desired computation. The transputer and the FPGA can then operate in parallel, exchanging data over the bus if necessary. The result in SRAM, if any, can be extracted at the end of the computation. Notice that since the FPGA and the transputer can run in parallel, the definition of $divide_n$, $conquer_n$ and $merge_n$ for the divide-and-conquer partitioning method can be modified such that the expression

$$\mathcal{S}\ (divide_n\ ;\ conquer_n\ [\mathcal{H}\ R,\ S]\ ;\ merge_n)$$

describes the concurrent execution of the software program S and the hardware implementing R.

A variant of occam, called Handel, can be used to pro-

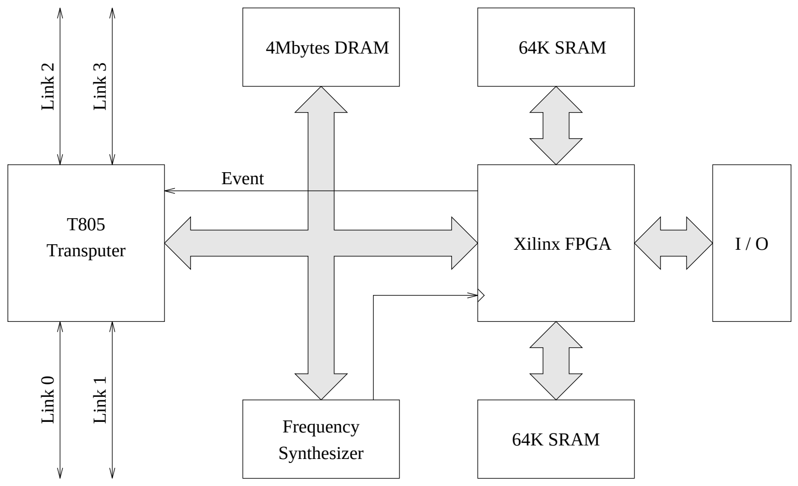


Figure 6 The HARP1 system.

gram the FPGA on HARP1 (see [14],[15]). There are library routines in Handel that provide the interface to SRAM and to the transputer bus for the FPGA, and it is possible to describe the core computation using Ruby and the interface using Handel. Hence our system-level compiler produces from a Ruby source description three target programs for HARP1: a Ruby program and a Handel program for the FPGA, and an occam2 program for the transputer.

Using our compilers and other tools, we have implemented a number of Ruby programs on HARP1, including sequence matchers and several butterfly networks for matrix transposition and for fast Fourier transform. The FPGA clock typically ranges between 5MHz and 20MHz, depending on the application. The implementation process is completely automatic – although for large designs the vendor software for placing and routing the FPGA may take a long time to complete.

8 Concluding remarks

Our declarative framework offers a number of advantages for parametrised hardware-software codesign, such as concise descriptions, flexibility of target and support for provably-correct development. It is particularly applicable to designs with a uniform structure, which can be found in, for instance, many signal processing systems. Implementations can be produced rapidly if the hardware is implemented by field-programmable devices.

While our experience of Ruby has been favourable, further research is required before realistic designs can be produced routinely. It would be useful to extend the expressive power of the system-level description language, perhaps by incorporating the stream model of Ruby (see [4], [10]) which provides additional primitives including delay operators, rate converters and abstractions for run-time configuration of components. Both the hardware and the software code generators of our experimental compilers can be optimised: part of this optimisation can be achieved using the algebra of Ruby.

We have also been investigating techniques for assessing the quality of hardware designs in Ruby [9]. Methods for estimating performance are being extended to take into account the effects of having multiple hardware and software units, and the overheads of communication between them. These techniques should facilitate the development of appropriate cost measures for mixed hardware-software systems, which can be used in algorithms and transformation systems for automatically optimising and partitioning designs.

Acknowledgements

The hardware Sobel edge detector was implemented by Steve Chiu. Thanks to members of the Oxford Hardware Compilation Research Group for discussions and suggestions, and to Vincent Lok and Geoffrey Randall for help in producing some of the diagrams in this paper. The support of Oxford Parallel Applications Centre, Esprit OMI/HORN project, Scottish Enterprise and Xilinx Development Corporation is gratefully acknowledged.

References

[1] Algotronix Ltd, *CHS2x4 Custom Computer User Manual*, 1992.

[2] R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice-Hall International, 1988.

[3] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.

[4] G. Jones and M. Sheeran, "Circuit design in Ruby," in *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990, pp. 13–70.

[5] G. Jones and M. Sheeran, *Collecting Butterflies*, Technical Monograph PRG–91, Oxford University Programming Research Group, February 1991.

[6] T. Kean and J. Gray, "Configurable hardware: two case studies of micro-grain computation," in *Systolic Array Processors*, J.V. McCanny, J. McWhirter and E.E. Swartzlander Jr. (eds.), Prentice Hall, 1989, pp. 310–319.

[7] S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[8] A. Lawrence, A. Kay, W. Luk, T. Nomura and I. Page, "Using reconfigurable hardware to speed up product development and performance," JFIT Conference, Edinburgh, March 1994.

[9] W. Luk, "Analysing parametrised designs by non-standard interpretation," in *Proc. International Conference on Application-Specific Array Processors*, S.Y. Kung, E. Swartzlander, J.A.B. Fortes and K.W. Przytula (eds.), IEEE Computer Society Press, 1990, pp. 133–144.

[10] W. Luk, "Systematic serialisation of array-based architectures," *Integration, the VLSI Journal*, Vol. 14, No. 3, February 1993, pp. 333-360.

[11] W. Luk and I. Page, "Parameterising designs for FPGAs," in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 284–295.

[12] W. Luk, V. Lok and I. Page, "Hardware acceleration of divide-and-conquer paradigms: a case study," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1993, pp. 192–201.

[13] W. Luk, T. Wu and I. Page, "Hardware-software codesign of multidimensional algorithms," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1994.

[14] 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.

[15] M. Spivey and I. Page, *How to program in Handel*, Technical Report, Oxford University Programming Research Group, December 1993.

[16] A. Wenban, J. O'Leary and G.M. Brown, "Codesign of communication protocols," *IEEE Computer*, Vol. 26, No. 12, pp. 46–52, December 1993.