Design Of Future Systems

Ian Page
Head of Hardware Compilation Research Group,
Computing Laboratory,
Oxford University, Oxford OX1 3QD, U.K.

Abstract

This paper describes a vision in which future systems consisting of novel hardware and software components are designed and implemented by a single type of professional engineer. That professional has more in common with today's programmer than a hardware designer, although both of these existing bodies of professionals have a strong contribution to make to understanding, defining and bringing about this transformation in product creation.

1 Design of Future Systems

Computing systems are becoming increasingly complex and often contain large amounts of both hardware and software. Currently, a yawning gap exists between the study and practice of computing, and the study and practice of digital electronic engineering. We believe that managing effectively the design of tomorrow's products necessitates the use of comprehensive design tools, methodologies, and education and training provision which can close this gap.

Errors in the design and understanding of systems usually congregate between the boundaries of sub-systems, particularly when the sub-systems have been engineered using different methods. This is certainly true when we consider systems with both hardware and software components. Historically it is easy to understand because the subject of computing was split at a very early stage in its development between the branches which deal with hardware and with software.

I take the view that a single style of system description is needed, from which both hardware and software sub-systems can be synthesised. I also believe that currently the only body of knowledge and practice which is capable of supporting the design of very complex systems is programming. It seems self-evident that the current styles of hardware description are completely inadequate to describe software. However, it seems to be the case that the languages, tools, and practices of software engineering *can* be adapted to meet the challenge of hardware design. Indeed, there has been an obvious historical process over the last twenty years in hardware design which has resulted in the ability to describe higher and higher levels of abstraction in hardware description languages. These have tended to look more and more like programming languages as time goes by. I believe the time is now upon us when we need to begin the completion of this evolutionary process by merging the two professions, also their tools and practices, and their educational and training support.

The work done in my own Hardware Compilation Research Group at Oxford University serves to highlight some important points. Our enabling 'act of faith' was to assume that at some point in the future we would all be routinely using programs as the single description of complete systems [4, 8]. Thus, we assume that many hardware-software systems will be created by programmers rather than mixed teams composed of programmers and electronic engineers. Since making this judgement, our work has been focussed on building the tools and methodologies to make this dream become a practical reality. Happily, I can report significant progress towards this goal, although there is still much interesting work to be done. Other articles giving a more detailed account of our work can be found in [6, 7, 9, 2].

For the most part, we concentrate on the tools and methods necessary to allow *programmers* to build working hardware-software systems. We believe that this stance makes our group rather different from some other groups who might use HDL descriptions to synthesise hardware. Another difference perhaps lies in the choice of languages and the style of programming. While there is much interest in synthesis from behavioural descriptions, we feel that purely behavioural descriptions do not capture enough information to allow synthesis of efficient hardware. Until the state of art in synthesis from arbitrary behavioural descriptions is good enough, we take the view that it is both reasonable and practical to ask a little more of the programmer.

The primary reason for implementing some system functions with hardware rather than with software is because of gains in speed of execution. Thus, as hardware is only fast because of its inherent parallelism, it is clear that the final hardware descriptions must exhibit an appropriate degree of parallelism. If the behavioural description does not contain, or strongly hint at, the appropriate parallelism, then it must be extracted automatically. Our experience is that this very difficult task is not, in general, yet within the grasp of automatic tools. However we have found that with appropriate languages and tools it is in fact quite reasonable to ask that programmers provide this level of description themselves. Crucial to this process is providing an abstract enough model of the descriptive language that the designer is not swamped by the myriad of details that an electronic engineer currently has to deal with.

We also believe that programming languages should be heavily influenced by mathematics. The notations used by designers need to be simple, appropriate, and highly trustworthy for us to have any hope that their designs will be right first time (or even last time!) and that they will be maintainable and reusable. Programming languages based in mathematics can help to ensure these qualities. In contrast, most existing 'real-world' programming languages do not even have their meaning properly defined in any document (i.e. they have no formal semantics), so there is simply no hope of building completely trustworthy systems on top of them.

In the short term at least, it may require a slightly new breed of programmer to describe and synthesise hardware-software systems; someone who successfully combines some of the skills associated with each of the two existing types of specialist. In particular these designer/programmers have to be aware of both the parallelism and temporal behaviour of their programs. The skills needed for dealing with time and space in a design to an intimate degree are perhaps currently exhibited rather more by digital engineers than programmers. However, we contend that it is not a difficult task to train someone to have both a good grasp of programming and a proper understanding of the temporal and spatial implications of their programs. We have experienced this many times in our own group when programmers have come into the group with a standard programming background and have quickly become competent at producing hardware implementations, sometimes in hours.

Moreover, we believe that a program can have the necessary expressive power to enable it to serve as an executable specification of a system, whether it be entirely in hardware, entirely in software (running on some off-the-shelf processor), or in a combination of the two. Moreover, we believe that programs are the *only* reasonable basis for a single framework which spans both hardware and software implementations. One justification for this is that very large computer programs are the most complex artefacts that human beings have developed to date and the paradigm of programming has clearly been at least adequate to this task. Apparently then, the tools and techniques of programming are already good enough to deal with the problems posed by large scale complexity in systems (although there are undoubtedly many improvements still to be made). Also, as no paradigm other than programming has yet emerged to deal with the software problem, so it seems natural to look to this paradigm to help with the design and synthesis of hardware also.

In our work, we use automatic program transformation within a CSP-style framework [3] to map a source program into a variety of target programs. CSP is a mathematical system for reasoning about parallel systems. It has an associated algebra which means that you can put together CSP programs with the same degree of confidence enjoyed when combining algebraic terms. There is simply no question that in putting two algebraic terms, say a and b together with the $+$ operator that the resulting term $a+b$ 'works'. Indeed, we expect and demand that $a+b$ 'works' no matter what any actual values of a and b might be now, or in the future.

Putting programs into an algebraic framework can, at best, have exactly the same benefit, so that combining two existing programs *must* have the expected behaviour. Only when we liberate programming from the current constraints of today's ill-defined and over-complex languages can we hope to enjoy this property. Only then will we be able to expect machines, as well as humans, to be able to combine existing programs together in new ways for us without reference to the programmers who originally wrote them or indeed any programmers at all.

In such a framework programs can be automatically transformed by computers into a wide variety of different forms. These different forms can all have different properties, such as the amount of parallelism that they exhibit or the amount of storage that they use. However all forms can share the same functionality as the original program. In other words, if the (automatic) transformation process sticks to the rules of the algebra, it is simply not possible for it to generate a program that doesn't do what was intended (i.e. the same as the program originally supplied).

By exploiting such a system, it is possible to transform user programs into forms which can be implemented directly as hardware, or as software, or as machine code for an application-specific processor together with a hardware description of the processor. Using such transformations, a single application program can be made to exploit the inherent, and widely differing, cost-performance characteristics that each of these forms has. Thus, hardware support can be given to the parts of the application that need it most.

2 Programs into Hardware

The Handel-C language has been developed at Oxford for describing programs which are (usually) compiled into hardware. Handel-C is a small subset of C, extended with the parallelism and communication constructs of Hoare's CSP and expressions of arbitrary bitwidth. The language is necessarily different from C itself because of the different nature of hardware and software implementations.

C is not adequate for our task because it is a sequential language. This means that algorithmic parallelism can't be expressed in C programs. However it is in the very nature of hardware implementations that they achieve their speed through exploiting parallelism. It is beyond the state of the art automatically to introduce the appropriate parallelism into a sequential program. For this reason we expect the programmer explicitly to denote the parallelism that is appropriate for the desired hardware implementation.

Programs, written in Handel-C, are mapped into hardware at the level of netlists by a series of transformations. This is documented in [6] but note that this paper does not use the Handel-C syntax. Fo the latest details of the Handel-C language see [5] or the web site of the company selling the compiler [1]. The implementations fully exploit the explicit parallelism of the programs themselves and serial execution is only introduced where explicitly required.

A feature of Handel which is of particular note is its simple model of time. By definition, each assignment statement takes exactly one clock cycle to execute and none of the other constructs in the language add any hidden clock cycles to the implementation. This regime gives control of the scheduling and allocation of the hardware resources to the programmer but in a way in which this added burden can be understood and handled.

If the programmer has explicit control over space (i.e. the gates of the implementation) and time (i.e. the clock cycles), then the language is by its very nature a hardware description language. Because they are a source of complexity, misunderstanding and delay, we avoid the problems of a designer worrying about such hardware-related concepts as gate count. Instead, we have them focus on time and give them a very simple model for dealing with the temporal behaviour of their programs. Since there is a natural tradeoff between space and time in most implementations, there is some rudimentary control over space but the programmer only has the much simpler concerns of counting clock cycles.

3 Actual Applications

To forestall a possible accusation of ivory-towered dreaming that the foregoing is not practical, I will briefly mention some of the applications that have been constructed by programmers using our Handel-C hardware compiler. For the most part these programmers have not been concerned at all with hardware and many have not understand hardware at all.

3.1 Video Games

We have had a lot of fun building a number of simple computer games on a single FPGA. We have found that we can get a single FPGA to perform all the functions of these games, namely video generation, the game itself, including all its memory, and the user interface as well. Note that this is without using any special graphics hardware or even a screen bitmap memory - the FPGA, a crystal oscillator, and a three D/A converters built only of cheap resistors is all there is! A standard VGA monitor shows the resulting video.

So far, we have so far constructed versions of tetris, space invaders, the game of life, breakout, and others. The space invaders game was noteworthy in that the programmer was an undergraduate student who knew nothing about Xilinx FPGAs, or indeed hardware and FPGAs at all. With no warning at all, he was given the task of implementing space-invaders in hardware and in just six hours on the first day he had produced a working hardware implementation of the core of the game on a single Xilinx 4005 chip. Further details of this work can be found in [10], reprinted in [11].

3.2 Image Warping

A more serious application was a real-time video image warping demonstration using Handel-C to target the 'Pamette' FPGA board from Digital Equipment Corporation. In it, a two-dimensional spatial mesh describes a source-to-destination warping and a video stream is passed through this mapping and displayed on a screen in real time. The implementation allows the mesh to be manipulated interactively using mouse dragging operations. The warping algorithm used is based on George Wolberg's implementation of Douglas Smythe's 'Two Pass Mesh Warping Algorithm'. In truth, it took rather a long time to coax the brand new, and largely unsupported, DEC Pamette board to talk to the PCI bus and to get its five FPGAs to talk to each other reliably, but the warping application itself was working with under two days of programming effort.

3.3 Positron-electron Collider

The Japanese High Energy Physics Laboratory is building a positron-electron collider to evaluate some of the design considerations for a proposed large linear collider. The beam path has to be accurate to a few tens of microns. Thus it is important to remove the effects of traffic-induced vibration and other earth tremors on the apparatus.

The damping ring uses 36 FPGA-based position controllers to implement dynamic stabilisation of the beam path via the 36 movable alignment tables carrying the beam steering magnets. Laser position sensing is used between neighbouring tables to determine location in each of the five degrees of freedom.

It was very pleasing that within four weeks, the engineer concerned had interfaced our reconfigurable hardware to the motion controllers, had written Handel programs to implement the control algorithm and had the whole system working. The unsolicited comment of this hardware engineer that "after two or three days, I completely forgot that it was hardware that I was developing" I found especially welcome.

An unexpected benefit of this collaboration was that when the system became operational, it was found that a design assumption for the ring was invalid. It had been assumed that the massive concrete foundations for the 50m ring would remain at a constant temperature. In practice, it was found that the planned strategy for controlling the beam could not work with the expansion and contraction actually found in the concrete platform. This necessitated a complete rethink of the control strategy. As the implementation was in Handel implemented in FPGAs however, the whole system was so flexible that only a very small amount of programming effort was necessary to overcome a problem which might otherwise have been massive.

3.4 Self-validating Sensors

We have been collaborating with the Oxford Control Engineering Group to build 'self-validating sensors' [12]. These are sensor systems which give an on-line guarantee of their accuracy. The Coriolis mass flow meter is an example of one of our more complex sensor systems. Its implementation uses three RISC processors, eight FPGAs and 5000 lines of Handel code.

The flexibility given by the Handel + FPGA implementation route has meant that many more experimental versions of the sensor could be built. Again unexpectedly, this has resulted in a huge increase in the accuracy of the sensor by using more sophisticated processing than originally envisaged. These experiments simply would not have been done if conventional methods had been used to implement the hardware.

This particular sensor has been very successful and a large American industrial partner is currently replicating it for extended field trials.

4 Conclusions

We have argued the practicality of representing the hardware and software components of a system within a single framework. With this framework, implementations of programs can be generated which have amounts of software and hardware corresponding with desired cost-performance characteristics. We believe that using a framework like this will eventually result in designs being provably correct, and that moreover such proofs will be a natural side-effect of the compilation process, not something that the designer has to do separately.

Although the current hardware and software languages that we use are actually different, they are at least close in spirit. It is our intention to combine them both into a single language when time and our understanding permit. The languages and support tools we have developed have been used by many different programmers to develop moderately complex hardware/software systems. We claim that this justifies our original 'act of faith' and that the future of much hardware/software design will be done by a process of software engineering.

Much exciting work remains to be done. For example: the design of a single language for hardware-software co-design, the design and implementation of hardware and software compilers for the languages, optimisation of hardware implementations, asynchronous hardware implementations, automatic design and implementation of microprocessors, knowledge-driven and language-driven placement strategies for silicon implementations, new architectures for FPGAs and much more.

It seems likely to us that the combination of high-level, programming-based approaches to hardware-software codesign together with reconfigurable hardware will fundamentally change the ways in which the digital systems of the future are designed and built.

If true, this has far-reaching implications for the necessary skills basis that industry must adopt and nurture to be successful in the future. Our universities will have to start breaking down some of the barriers that exist between the subjects they currently teach. Hardware engineers, who already have an excellent understanding of the nature of parallelism, will have to become more programming-literate, and programmers will have to become more aware of parallelism and the time and space implications of the programs that they write.

Acknowledgements

I wish to thank all the members of my Hardware Compilation Research Group who make it worth coming into work each day - even when I'm on sabbatical leave! Their support has been invaluable, inspirational and hugely enjoyable. I would also like to acknowledge the work of Timo Korhonen who implemented the collider control system.

References

[1] Embedded Solutions home page. http://www.embedded-solutions.ltd.uk/.

[2] Jonathan Bowen, Jifeng He, and Ian Page. Hardware compilation. In J.P. Bowen, editor, *Towards Verified Systems*, Real-time Safety-Critical Systems, chapter 10, pages 193–207. Elsevier Science B.V., Amsterdam, 1994.

[3] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[4] C.A.R. Hoare and Ian Page. Hardware and software : The closing gap. *Transputer Communications*, 2(2):69–90, June 1994.

[5] Embedded Solutions Ltd. Handel-C reference manual.

[6] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[7] I. Page. Reconfigurable processor architectures. *Microprocessors and Microsystems*, May 1996. Special Issue on Codesign.

[8] I. Page. Towards a common framework for hardware and software (keynote paper). In M. E. de Lima, editor, *IX Brazilian Symposium on Integrated Circuits.* Brazilian Computing Society, Sociedade Brasileira de Computação, March 1996.

[9] Ian Page. Automatic design and implementation of microprocessors. In *Proceedings of WoTUG-17*, pages 190–204, Amsterdam, April 1994. IOS Press. ISBN 90-5199-1630.

[10] Ian Page. Compiling video algorithms into hardware. In *Advice97.* EDA Ltd, London, Jul 1997.

[11] Ian Page. Compiling video algorithms into hardware. *Embedded System Engineering*, Sep 1997. Reprint of Advice97 paper.

[12] Ian Page. Hardware compilation, configurable platforms and asics for self-validating sensors. In M. Glesner W. Luk, P.Y.K. Cheung, editor, *FPL97*, number 1304 in Lecture Notes in Computer Science. Springer Verlag, 1997.