Our Future Engineers Can Bridge the Software/Hardware Paradigm Chasm

Dirk Stroobandt

Electronics and Information Systems Department, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
email: Dirk.Stroobandt@elis.UGent.be

Abstract

Traditionally, hardware designers and software developers are educated in totally different ways, which is already prevalent in the use of the words "designers" and "developers." The claim in this paper is that we have to remove this distinction and bridge the chasm between the two worlds. This is necessary because current systems require a mixture of software and hardware design to fully embrace the possibilities offered in modern chip technologies. On the other hand, modern technologies are experiencing severe problems that effectively will halt Moore's law in the way we are used to it. Technology scaling no longer offers clock speed improvements and the microprocessor world is rapidly moving towards multi-core systems. This trend actually offers new opportunities for unifying the hardware and software paradigms as exploring fine grain, low level parallelism will be the name of the game in both worlds. In this unification process, reconfigurable systems may play a significant role.

1. INTRODUCTION

Most computer science (CS) departments at major universities originally were split from a much older electronics engineering (EE) department. The enormous boost in computer science research brought by microelectronics industry and for sure the introduction of the Personal Computer rightfully stirred a movement toward the separate discipline of computer science. Some schools, however, decided to keep the two disciplines tied closer together in an Electronics and Computer Science (ECS) Department. Nowadays, this actually turns out to be a very interesting constellation for cross-discipline research and teaching.

Whatever the administrative subdivision, fact is that the clear division between hardware (ASIC) design (traditionally more prevalent in EE departments) and software design no longer holds. Current systems are a mixture of the two. There are several reasons for this evolution:

- Moore's law has enabled ever more powerful systems on the same die area. The doubling of the number of transistors per unit area every technology generation allowed designers to put more functionality on a single chip, even to the amount that it is no longer manageable. This leads to the infamous *design gap* as the number of designers or the time needed for a large design can no longer keep up with Moore's law. In order to further improve productivity, the only solution is to reuse big existing designs and combining these. This is called IP (Intellectual Property) Reuse. Where this combination of "chips" to a complete system used to be done at the board level, it is now possible at the chip level, leading to a System-on-Chip (SoC) design methodology. In such a SoC, scheduling and arbitration are becoming more important, bringing software issues to the hardware designer's world.

- In microprocessor architectures, Moore's law has often been translated as "the clock speed of the system doubles with technology generation." This was true until the end of the 1990's but then came the time when the interconnect delay started dominating the overall delay in chips (Figure 1). Indeed, while transistors get faster when they are scaled down in size, the interconnects between them get slower as both the resistance and capac-

This paper is based on experiences from teaching two courses at Ghent University, Belgium: "Design Methodology for Complex Systems" and "Hardware/software Co-design."

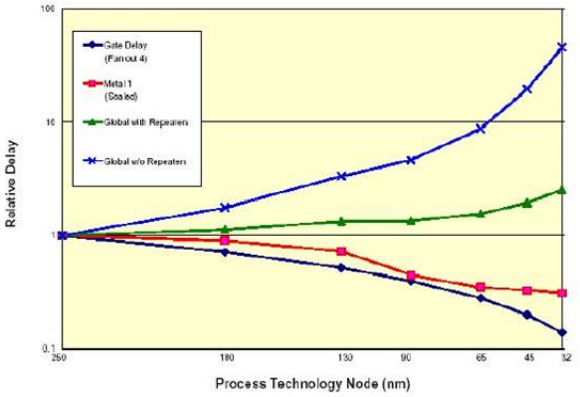

Figure 1. Interconnect delay inversely scales with technology generations thus leading to a halt to processor speedup by simply scaling the technology (figure courtesy of Dennis Sylvester).

itance of the wire increase. Once the wire delay starts to dominate, one can no longer thrust on scaling alone to improve microprocessor speed. On top of that, the scaling itself is slowing down and there are so many technological barriers to further scaling that many researchers predict the end of processor speed improvement really soon. Microprocessor companies therefore lay all their eggs in the basket of multi-core systems. To obtain sufficient performance improvement, hardware accelerators are getting more and more a necessity.

- With the increase in compute power, also the applications are getting more demanding. Ubiquitous computing is becoming the name of the game and people are starting to expect access to high quality multimedia data (especially video) everywhere. This puts an enormous pressure on multimedia hardware systems and requires the use of specialised architectures and the exploitation of massive parallelism. Again, hardware acceleration is needed and hardware blocks need to coexist with a software oriented environment.

- The advent of reconfigurable hardware (FPGAs) has enabled much of the above trends as it has provided a much more flexible hardware infrastructure that can co-exist and cooperate with a software architecture more easily. At the same time, the very simple FPGA architectures of the past (containing an array of similar simple cells) has evolved into much more hybrid arrays of logic blocks, memory blocks, multipliers and other DSP blocks, making the FPGA more efficient for the complex tasks it is required to do. The introduction of first soft-cores and later full-fledged hardware computing cores on the FPGA chips really brought hardware/software co-design to the heart of reconfigurable systems.

Given the strong link between hardware and software design in today's systems, university education should focus on hardware/software co-design and train new electrical engineers and computer scientists in this new cross-disciplinary domain. In this paper, I will present the first steps towards such education at Ghent University in Section 2. Section 3 will then focus on the main issues that I deem important in designing a system: exploring the trade-offs between all the implementation choices the ever expanding design space has to offer. As explained in Section 4, the main driving force behind combining hardware and software design methodologies (and courses) is the need to find and exploit low-level parallelism. With that, and Section 5, I will conclude.

2. TEACHING HARDWARE/SOFTWARE CO-DESIGN IN PRACTICE

At Ghent University in Belgium, we have adopted the Bachelor/Master structure which resulted from the Bologna/Sorbonne meetings that changed much of European education. All over Europe, a 3-2-3-structure will be implemented: 3 years of Bachelor studies, 2 years for the Masters degree and (not yet implemented) 3 years for obtaining a PhD. The change from a 2+3 structure to this new structure was the ideal time to evaluate our existing curricula and propose some changes. In the Faculty of Engineering at Ghent University these changes were rather fundamental and the entire curriculum was set up from scratch. The basic ideas behind our engineering studies remained however:

1. A strong focus in the Bachelor on the mathematical background needed in the Masters.

2. A focus on learning to reason and solving problems unknown before, as well as to "think out of the box."

3. A broad base of domains rather than a narrow focus.

Within these boundaries, several proposals were made to reorder the study trajectories. At some point, the idea was to base the trajectories on application domains such as ICT, Multimedia and Embedded Systems but in the

end pretty much the old structural divide between "electronics" and "computer science" prevailed. However, within these trajectories, several options are offered (as main subjects):

- For the Master of Electrical Engineering:
  - Main Subject: Electronic Circuits and Systems
  - Main Subject: Information and Communication Technology
- For the Master of Computer Science Engineering:
  - Main Subject: Software Engineering
  - Main Subject: Information and Communication Technology
  - Main Subject: Embedded Systems

The Master of Electrical Engineering contains the two classical domains of Circuits and Systems and ICT, both of which have a strong industrial relevance in Belgium. These are more hardware-oriented and also contain courses on the technology and design of computer chips and ASICs. In the Master of Computer Science Engineering, however, the Main Subjects are more diverse. The Software Engineering option is really focused on managing large software projects. There is also an ICT option which is more focused on network traffic and the software side of ICT than in the electronics master. The real new option here is the one on Embedded Systems which was specifically targeted at the hardware/software interface. This was intended to address the need for a more modern education at the forefront of the technical evolutions. Unfortunately, as this option was formed within the Master on Computer Science, it inherited all of the main software courses that are common for all computer science options. Hence, courses also include Design of Distributed Software, Software Architecture, Queueing Theory, Information Theory, Design of Multimedia Applications and Multimedia Networks. Only Advanced Computer Architecture, Compilers, Complex Systems Design Methodology, and Hardware/Software Codesign are courses that are really on target. This has fielded some complaints by students who specifically chose this option to learn more about the hardware issues, only to find that two-thirds of the main courses are still software-oriented. For this reason, a re-evaluation of the Embedded Systems option is being considered.

I strongly believe that a major issue in any embedded systems program is to find the right balance between hardware and software. As we already have a stronger software side in the courses at Ghent University, I tend to focus on the hardware part much more in the two courses I teach: Complex Systems Design Methodology and Hardware/Software Codesign. The first course focuses more on the inital system design steps of specification, architecture evaluation and hardware/software partitioning (see Figure 2), while the latter course really targets the hardware design methodology as well as some specific hardware/software issues such as the use of real time operating systems.

For completeness, I list the chapters addressed in both courses I teach:

- Complex Systems Design Methodology
  1. Embedded systems, System-on-Chip and platform-based design
  2. System specification techniques
  3. Architectures for complex systems
  4. Exploring the design space
  5. The importance of interconnects
  6. The importance of embedded memory
  7. Predicting performance
  8. Interfaces and interface design
- Hardware/Software Codesign
  1. Design methodology and architectures
  2. High level VLSI Design
  3. Logic Synthesis and Physical Design
  4. Power management on circuit level
  5. Testing of digital systems
  6. RTOS
  7. Code optimization for embedded systems
  8. Power management at system level
  9. System-level testing

Although the course on Hardware/software co-design has grown from an earlier course on VLSI Design, the focus now is much more on reconfigurable design, especially in the labs.

Labs are more important than theory in any design course. Indeed, students can only learn the intricacies of hardware design by hands-on experience. Hence, we specifically paid attention to setting up enough labs for the students. With the much appreciated help from my Ph.D. students, we have opted for closely guided labs in which all the system design steps get attention. Especially in the Hardware/software co-design course, we go through the different steps of designing a concrete system, in this case a Viterbi coder. This is synthesized for real FPGA hardware.

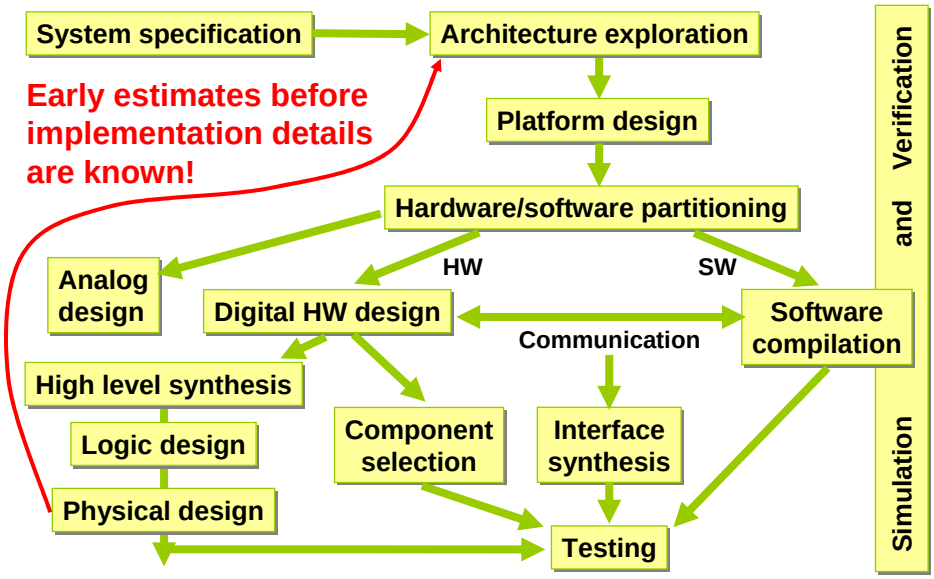

Figure 2. The design methodology for designing complex systems.

The main purpose of both courses is to introduce a different way of reasoning about problems to software-trained students. The hardware approach requires a more structural view on a problem, emphasizing the inherent parallelism. In the majority of courses, however, students have only been confronted with the procedural (algorithmic) way of thinking. Even though object-oriented programming and aspect-oriented programming have found their way in the courses, these are not focused on parallelism in the first place. For most computer science students, this structural view is completely new. I will come back to this issue in Section 4.

3. IT'S ALL ABOUT TRADE-OFFS

One of the things I feel students should learn is that there is no single embedded systems design methodology. Desiging always is a matter of making trade-offs. Depending on the problem at hand, the performance features wanted and the available options in the design space, a completely different solution might be suggested for two designs that are intrinsically very similar. This may seem "natural" to people designing systems, it makes teaching system design very difficult. Also in teaching, choices have to be made on what to teach and how. In the last three years, I have tended towards trying to give as broad an overview as possible, spending a lot of time listing, and explaining about, various problems and their multitude of solution options. Yet, this is a bit boring, and not only for the students. So I believe a good practical example is needed to show in more detail how designing could be done. But what is a *good* example? I am still looking for one. Also, as time is limited, explaining such an example in detail invariably means there is less time left for exploring other options, thus limiting the scope of the course. Again, there is a trade-off to be made here.

In teaching about the trade-offs, a number of issues are important to address:

- One of the most important aspects in making trade-offs, is the choice of performance measures. A good trade-off can only be made when the right questions are asked and the right objectives are set. Any course on embedded system design must therefore, in my opinion, spend enough time on explaining what performance measures are most important today (power, cost, time-to-market, latency, bandwidth, and area are the major ones). Not only should students know why these are important, also the impact of current technologies on these performance measures is important.

- One of the critical issues is the design level at which trade-offs are made. In the first design stages, the design description is very abstract. Yet, one immeadiately is confronted with very important design decisions on the architecture of the system. In this architecture exploration step, not

a lot is known about the details of the final implementation (as the idea is exactly to abstract most of this away). This leaves a lot of implementation choices for later but it also means it is hard to say something about the relative performance of the solutions to choose from. Therefore, very early high-level performance estimations are paramount. The estimates at this level will not be very accurate (how could they be when almost nothing is known in detail yet?) but they should give a good relative appreciation of the solutions in order to retain the right architectures. In my view, this is relatively uncharted terrain where more research would be welcome. The estimations also changed a lot over time as almost all performance parameters nowadays are dependent on interconnect parameters rather than the computing elements themselves. Interconnect estimation hence became an active field of research since the end of the 1990's [1, 2, 3, 4].

- After (and sometimes during) architecture exploration, one also has to decide which parts of the problem will be handled by software and which parts should be taken care of by a hardware component. This hardware/software partitioning again induces a very important trade-off, that of flexibility versus computing strength (time needed to perform a certain task). Although some people believe there is nothing much to gain in this area, I still have to see the first software/hardware partitioning tool that can automatically decide on the hardware/software boundary without the need for extensive simulation. I think there are opportunities left for research here. Also, not all issues have been taken into account in the past. Time to complete a computation always has been an important factor (and still is) but the partitioning tools should also consider power, area, cost, etc.

- Another aspect in architecture exploration is the trade-off between flexibility and performance. The full performance (in terms of speed, power, area, etc.) can only be obtained with custom-made hardware (an ASIC) but this comes at a high cost as the hardware is fixed for a single task and there is no flexibility to change tasks. At the other extreme, using a classical microprocessor offers full flexibility as this is programmed and can basically perform all (algorithmic) tasks. However, the processor is optimized to perform well for average tasks, not for the specifc one at hand. FPGAs fill a whole lot of the space in between these two extremes. Their (re)configurablity still leaves enough flexibility while the hardware-oriented design approach actually allows a good performance. With dedicated hardware blocks (such as multipliers) the performance of FPGAs is improved significantly and the introduction of soft and hard cores on the FPGAs enables a fully flexible programmable solution as well. The choice for a reconfigurable component therefore still leaves a lot of implementation options on the table. This makes reconfigurable platforms ideally suited not only for designing systems, but especially for educational purposes.

4. THE FORGOTTEN PARADIGM: PARALLELISM

There used to be a big chasm between the way in which computers were programmed (in an algorithmic way, using procedural languages) and the way hardware was designed (structurally, using parallelism and hardware description languages – HDLs). Software engineers and hardware engineers therefore speak a different language. Of course it is true that there have been many attempts to introduce more parallelism in computers (remember the transputers of 30 years ago) and in software. But these attempts still stay intrinsically sequential in nature and the limited amounts of course-grain parallelism found today is nothing compared to the massive use of fine-grain parallel structures in hardware design. However, with the unavoidable trend towards multi-core systems, also software engineering will have to embrace low-level, fine-grain parallelism much more tightly. One may argue that parallel computation is not the holy grail (remember Amdahl's law?) but then again the amount of parallel processors we are talking about today is multiples of what people could dream of earlier. Hence the parallelism we should exploit in software projects may no longer be the thread-level parallelism but rather the massive low-level parallelism that is also exploited in hardware design. This is a very interesting evolution as it actually conjoins both paradigms into a single one. Hence, in the future, there simply should no longer be a chasm between software and hardware engineers!

This likely does not make things easier. Programmers are still used to procedural thinking. A paradigm shift as described above will (and should) not occur overnight. But if we want our students to be ready for tomorrow's challenges, I believe we should train them in exploiting low-level parallelism structurally, whether they major in computer science or hardware design. And it is not that such a shift hasn't happened before. Object-oriented programming is getting worldwide acceptance because it is the only way in which large software pro-

jects can be managed. This programming paradigm is based on modularity (which includes a kind of parallelism). But the goal was not to introduce parallelism so it will not suffice in our new world. Also aspect-oriented programming (even newer and less adopted) implies some kind of parallelism but again the main concern here is the separation of concerns, not the parallelization itself.

Although the main focus should be on the concepts (including finding and manipulating Data Flow Graphs), a big concern in the massively parallel paradigm is the design language to be used. HDLs have much better facilities for describing parallel behaviour but they lack the benefits of modern software languages (such as object orientation, refactoring facilities, etc.). The design community still has not decided on what language is best fit for describing both software and hardware (is it SystemC, SystemVerilog, ...?). The requirement to exploit low-level parallelism adds another dimension to this discussion. Fact is that the designer should be releaved from as much burden as possible. Once a good description framework is found, tools will be needed to (semi-)automatically translate this description in multicore software or in a hardware description to be implemented. A few examples of such tools are being developed [5, 6, 7] at various institutions. There is a lot of research left here and it is hard to include the newest research results in teaching. But a good start would be to change the fundamental way of thinking towards a structural massively parallel paradigm. And this can be done right now. In this respect, it is interesting to note that papers and classes at the last Embedded Systems Conference (ESC) (April 2007) suggest that hardware and software development are practically one and the same in terms of code generation.

I believe the role of Reconfigurable Computing is not to be underestimated in this paradigm shift. In order to get a "feeling" about the low-level parallelism, students will have to test their gained knowledge on practical examples wich can easily be done on FPGA hardware.

5. CONCLUSION

Hardware designers are used to thinking about systems structurally, with parallel flows of information handling.

Software developers are more acquainted with a procedural view and the sequential execution of instructions. However, with the strong drive toward multicore systems (in the near future with hundreds of cores), massive parallelism will also be the main design goal in developing software. Hence both worlds will actually grow towards using the very same paradigm. It is therefore high time we bridge the chasm between hardware design and software development and start educating our electrical engineering and computer science engineering students with the same basics.

References

[1] D. Stroobandt, *A Priori Wire Length Estimates for Digital Design*. Boston / Dordrecht / London: Kluwer Academic Publishers, 2001.

[2] D. Stroobandt, "Tutorial: A priori system-level interconnect prediction: Rent's rule and wire length distribution models," in *Proceedings of the 2001 International Workshop on System-Level Interconnect Prediction*, P. Christie, J. Davis, and D. Sylvester, Eds. Rohnert Park: ACM Press, 4 2001, pp. 3–21.

[3] D. Stroobandt, "Guest editorial : System-level interconnect prediction," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 10, no. 2, p. 175, 4 2002.

[4] D. Stroobandt, "Recent advances in system-level interconnect prediction," *IEEE Circuits and Systems Society Newsletter*, vol. 11, no. 4, pp. 1; 4–20; 48, 2000.

[5] P. Faes, M. Christiaens, and D. Stroobandt, "Mobility of data in distributed hybrid computing systems," in *Proceedings of the 21st International Parallel and Distributed*. Long Beach, CA, USA: IEEE Computer Society, 1 2007, p. op CD.

[6] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, and D. Stroobandt, "From loop transformation to hardware generation," in *Proceedings of the 17th ProRISC Workshop*, Veldhoven, 11 2006, pp. 249–255.

[7] P. Faes, M. Christiaens, D. Buytaert, and D. Stroobandt, "FPGA-aware garbage collection in Java," in *2005 International Conference on Field Programmable Logic and Applications (FPL)*, T. Rissa, S. Wilton, and P. Leong, Eds. Tampere, Finland: IEEE, 1 2005, pp. 675–680.