The Exponential Impact of Creativity in Computer Engineering Education

Patrick Schaumont
schaum@vt.edu

Ingrid Verbauwhede
ingrid.verbauwhede@esat.kuleuven.be

***Abstract*—Embedded computers continue to show a relentless pace of improvement with better technologies at greater levels of integration for ever broader application domains. Furthermore, access to such technology keeps on becoming easier and cheaper: design kits, design examples, data sheets and application notes all are just a mouse-click away. This, however, did not make learning and education of computer engineering techniques any easier: the basics of computer engineering remain the same, but the tower of complexity keeps growing higher. This paper highlights *creativity* as a fundamental quality that computer engineering students need to cultivate to master modern technology. We propose *open-ended design practice* as an essential tool to nurture creativity. We report on the results and lessons learned from an embedded-systems design course which we organized for the past 7 years. An important observation is that creativity has an exponential impact on the quality of design delivered by students, when quality is measured by factors such as performance, power, and cost. This means that the best students make designs that are literally orders of magnitude better than those from weaker students, even for students in the same class and the same curriculum. In this paper, we do not intend to explain the causes of this difference. Rather, we analyze what can be done with creativity in a classroom setting, and we provide examples from our past courses.**

***Keywords*-Design Practice, Design Project, Computer Engineering Education**

I. Introduction

It's well known that computer engineering is a field with a very high level of innovation. It is fueled by rapid technological progress. Modern information technology completely flattens the access to know-how and technical reference data. Students face a deluge of information and facts, and they have an unprecedented amount of tools and design support at their disposal. Educators face the problem of creating courses that show students how computer engineering is a well-structured, well organized field with solid and well-connected study subjects.

This a challenge. Computer engineering misses a comprehensive basis of knowledge. There are well-established subdomains, such as hardware design, computer architecture, or operating system design, but the walls between the subdomains are high. Furthermore, there is a continuous and rapid technological evolution within each subdomain. The results reported in this paper span a period of 7 years; they cover multiple generations of tools, design kits and technologies. The changes are more than cosmetic: they involve changing the design libraries and sometimes the design languages. In the next seven years, it's very likely that the prototyping platform will change several more times. The status quo is unacceptable; industry expects graduates that are up to date with the latest technologies.

This paper addresses the problem of educating computer engineers who are able to master continuous change: change in the technology, change in the tools, change in the design methods, change in the applications. We advocate that creativity is a fundamental quality to help students to tackle change. Creativity is the use of imagination to create something (Oxford Dictionary). To be creative, an engineer has to be willing to step out of the box. The engineer has to combine known facts and concepts into new, innovative results. T. Wagner puts it as follows: *"Today, because knowledge is available on every Internet-connected device, what you know matters far less than what you can do with what you know."* [4]

This is especially true in the context of engineering, which emphasizes the useful application of academic knowledge. We don't think of creativity as an artistic (or magic) quality that is born onto someone. Instead, creativity can be cultivated by practice and experience. Creativity, like teamwork, is a soft subject, with a high 'experience' factor and a low 'theory' factor. Design practice, or learning by doing, is one of the more meaningful ways to encourage creativity and to quantify its impact. This can be done by structuring courses around open-ended design projects. *Open-ended* means that these design projects do not have a single, fixed solution, but rather a solution space. It is up to the students to explore and uncover the solution space. The paper will describe several such examples taken from previous courses.

Many computer engineering curriculum treat design projects as a capstone, a finishing touch in an education that brings everything together. This is good, but design practice and open-ended design does not have to be limited to capstones. Design practice is a valuable goal in itself. This argument is certainly not new, and it has been applied to engineering education [1]. However, we noted that complex computer engineering problems seem to increase its importance over the years.

A surprising result from the open-ended design projects is that the design quality delivered by students varies by *orders of magnitude*. In this case, we measure design quality using traditional metrics such as power, area, and performance. These variations in student performance persist regardless

of the type of application problem, or the student body, or the design environment.

Creativity thus has an exponential impact on design quality. This is a dramatic result. It means that, in a class of 100 students, there is one student who produces a solution which is 10 times better than that of 9 other students, who in turn produce results that are 10 times better than those of the remaining 90 students. This introduces a challenge for the educator. Should we worry about those 90 students who failed in delivering a better result? Or should we praise the top student as the class genius that makes the educational effort worthwhile? Our viewpoint is that all of it matters. The educational effort should go to discuss, with the whole class, the difference between failing and winning designs. This also allows us to qualify the many dimensions of design quality (the smallest design, the fastest design, the most flexible design, ..), and to point out how a single problem has many different solutions.

In the remainder of this paper, we provide several examples of design projects in the hardware/software codesign problem space, along with student performance data for those projects. Next, we describe practical approaches at organizing design projects in a manner amendable to such a course format.

II. The Codesign Challenge

Each year, the senior-level Hardware/Software Codesign course at Virginia Tech organizes the *Codesign Challenge*. The project assignment for students taking this course is to improve the performance of a reference design on an FPGA design kit as much as possible using Hardware/Software Codesign techniques. The reference design is typically a computation-intensive application written in C on a soft-core processor. The acceleration techniques applied by the students include a broad range of codesign techniques: hardware acceleration using memory-mapped coprocessors, custom-instruction set design on the soft-core, task-level parallelization through allocation of multiple soft-cores, customization of the communication- and storage infrastructure. Many, but not all, of these techniques are discussed in the course lectures; the course relies on a text-book to provide a structured introduction to architecture specialization and low-level software optimization [2].

The project comes as the final assignment of the course, and students have approximately two weeks to design a solution. As an open-ended assignment, there is no predetermined solution. Instead, students are told that all designs will be strictly ranked according to performance (as measured with a pre-defined, application-dependent testbench), and that their rank will determine their grade. To deal with closeties, two additional ranking criteria are applied.

1) If the performance of two designs is within 5%, then the design area (typically measured as FPGA utilization in LUTs, LEs, registers, RAM cells, etc) is used as a secondary criteria. Smaller designs are of course better.
2) If the performance AND the area of two designs is within 5%, then the design turn-in time is used as the final criteria. Earlier turn-in is better.

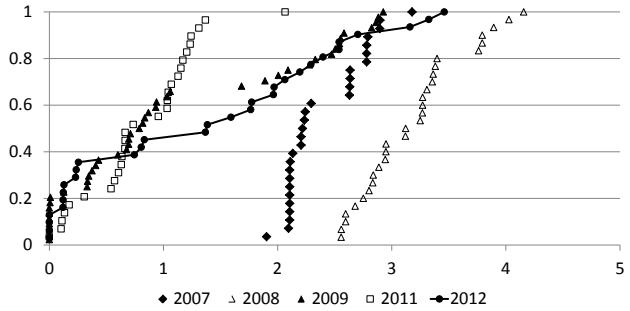

Figure 1. Distribution of results for 5 projects. The X-axis is the log_{10} speedup over the reference implementation, the Y-axis enumerates the student population.

The assignment changes each year. For example, the course has used a CORDIC application (2007), an array bit-counting application (2008), a 256-bit data multiply-accumulator (2009), a Fast Fourier Transform (2010), a bit-matrix multiply (2011) and a Mandelbrot Fractal Generator (2012). For brevity, the applications will not be discussed in detail, but we provide the assignments online (at http://rijndael.ece.vt.edu/challenge). In every year, apart from 2010, students were able to obtain several orders of magnitude of performance improvement over the reference design. The 1024-point FFT application from 2010 turned out to be too complex to handle in just two weeks, and the students did not achieve a decent result. In the following, we will therefore not include the results from the 2010 project.

Figure 1 illustrates the class performance as a population in function of speedup achieved for 5 different projects. Several aspects from this graph are striking. Regardless of the application, there is a performance difference of two to three orders in magnitude between the best and the worst result. While the worst results in a project hardly include any optimization (beyond eg. trivial compiler optimization), the best results are extremely good. In fact, the best results seem to be competitive with what an experienced designer in industry could achieve. The best designs typically not only identify mechanisms for acceleration of the computational bottleneck, but they also provide a comprehensive analysis of system bottlenecks. Such designs have an optimized communication infrastructure, an optimized storage hierarchy, and they achieve optimal parallelism at system-level.

A second striking feature from Figure 1 is that results are clustered: the distribution is not smooth but shows sudden jumps in improvement of the design performance. This clustering effect originates from optimization techniques, which typically have a large, non-linear impact. Optimiza-

```
x0 = scaled x coordinate of pixel
y0 = scaled y coordinate of pixel
max_iteration = 30
x = 0
y = 0
iteration = 0
while ( x*x + y*y < 2*2 AND
        iteration < max_iteration ) {
  xtemp = x*x - y*y + x0
  y = 2*x*y + y0
  x = xtemp
  iteration = iteration + 1
}
color = iteration % colormap;
plot(x0,y0,color)
```

Figure 2. Mandelbrot pixel drawing routine

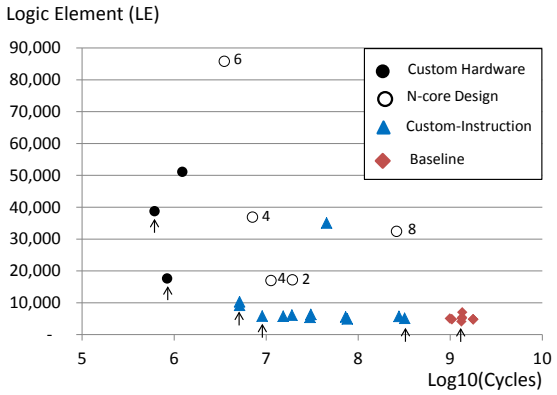

Figure 3. Time/Area trade-off for various architectural styles, including software optimization (baseline), custom-instruction, multi-core, and full-hardware designs

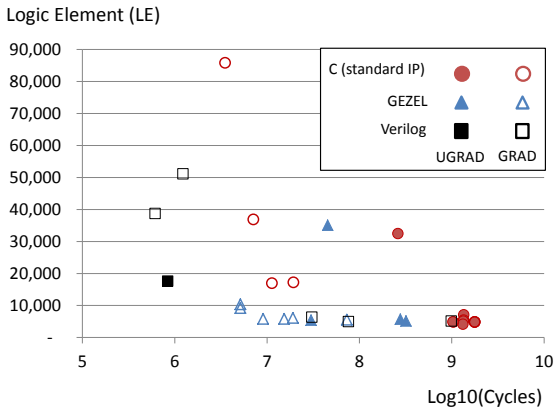

Figure 4. Time/Area trade-off for the designer population, classified by level (graduate/undergraduate) and design tool used (standard IP/ GEZEL/ Verilog)

tion in this type of project frequently means adapting the implementation at different levels of abstraction in hardware and software. For example, a student may explore the use of compiler optimization in software, and then realize that the memory hierarchy of the system needs to be adjusted (with cache, on-chip memory, and so on). Or, a student may be evaluating a coprocessor design, and then realize that the data handled by the coprocessor causes a communication bottleneck between the software and the hardware coprocessor. This needs to be addressed by revising the hardware/software communication architecture. These optimizations require 'thinking outside of the box'; and they tend to cluster students in groups according to their ability to do so.

III. An Example: Mandelbrot Fractal Project

To clarify why there is so a large difference between the results obtained by students, we will discuss the 2012 project, a Mandelbrot Fractal Generator, in further detail. The original design for this application is based on Chu's book [3]. The objective of the project is to display the well-known Mandelbrot Fractal on a VGA screen. Figure 2 shows the pseudocode to compute the display color for a single pixel; the listing is shown to provide an appreciation of the computational complexity. There are three data-multiplies in each iteration, with up to 30 iterations per pixel. As there are 480*640 pixels in a VGA screen, a full frame of the fractal requires over 27 million data-multiplies. To avoid the complexity of floating-point hardware, the reference C is written using a fixed-point data type with a precision of 28 fractional bits.

The reference architecture includes a 50MHz Nios-II/f processor (6-stage RISC pipe, 4K I-cache, 2K D-cache), a VGA controller, off-chip SRAM video buffer, off-chip DDR program memory. The reference testbench, which measures the time to draw one complete VGA frame, requires 1.775 billions clock cycles. The reference design uses a DE2-115 FPGA board with a large Cyclone IV FPGA. Figures 3 and 4 describe the solutions obtained by the students for this design, as a time/area trade-off. The reference design is at the lower-right corner of the figure, while the winning design is the leftmost point. The Y-axis is scaled in logic elements (LE), a unit of configurable space in the Altera FPGA. The figure does not reflect the usage of on-chip memory and dedicated on-chip DSP units. Figure 3 characterizes the architectural style used in the accelerator solutions in four major categories: designs with only software-optimization, custom-instruction designs, multi-core designs and full-hardware designs. The winning design generates the Mandelbrot fractal in real time. Since the VGA pixel rate is only 25MHz, this is achievable by unrolling the pixel drawing loop (Figure 2) in hardware and pipelining the result. This is, of course, a complex transformation that entails a complete rework of the system architecture. Multi-core solutions achieve a high performance as well. They come as a natural solution to the students because of the parallel nature of the Mandelbrot application: it's easy to partition the VGA screen in multiple areas that evaluate

in parallel. Multi-core solutions are, however, area-hungry, and they quickly lead to communication bottlenecks which cannot be easily diagnosed by the students. Using a custom-instruction design leads to a more incremental solution: students start by moving small parts of the pixel-drawing loop into hardware, up until the point where a complete pixel drawing loop executes as a variable-length custom-instruction.

Figure 4 makes an attempt at characterizing the designer population. The figure distinguishes graduate and senior-level students in the course, as well as the design tools/languages they have used. The circle symbol reflects designs that use standard IP modules available in the FPGA design environment. The triangle symbol reflects designs that use custom IP modules build on top of standard interfaces; the course uses the GEZEL design environment for this purpose. Finally, the square symbol reflects designs integrated at the RTL level in Verilog. The figure shows, as expected, that graduate students are usually more experienced and hence more adept at design space exploration. However, the figure also demonstrates that the best undergraduate students easily outperform graduate students. We have observed this effect to be consistent over many projects.

IV. Class Evaluation

When students are confronted with results as in Figure 3, they are thrilled, not disappointed. While they may regret for not having tried harder at the solution, they are engineers and hence have an ingrained respect for strong solutions. The instructor for the class makes a point of dedicating one lecture to the discussion of the results, and analyzing with the students why one solution is better than another.

A very effective approach is to start the lecture by discussing the concept of Pareto optimality. A design A is better than a design B if, in Figure 3, the design is both faster and smaller. When choosing a one-sided constraint in area or performance, it's always possible to pick one single design in the figure that will be Pareto-optimal. Several such designs are indicated by means of an arrow. The lecture then includes short impromptu presentations by students with a Pareto-optimal design. These presentations give a comprehensive overview of the design space.

A factor that further eases the students in not feeling intimidated, is that the project only carries one third of the total course grade, while two-thirds are allocated to more traditional evaluation mechanisms such as homework and quizzes.

V. Structuring Creativity

Finally, we provide some suggestions on structuring computer engineering courses in a way to encourage students to be creative. We believe these ideas are equally applicable to digital design courses that teach HDL, microprocessor engineering courses that cover interface techniques, or comprehensive 'embedded systems design' courses that address the complete design cycle. The course material is based on three components: design methods, examples, and design modeling/languages.

- **A foundation of design methods:** The central part of a computer engineering course is devoted to design methods. Similar to *design patterns* in software and architecture, design methods capture the essence of a solution and make it universally applicable. Unlike technology, design methods have a long shelf life. Prototyping kits may change every year, but design methods stay.
- **A stack of examples:** As creativity is so strongly practice-oriented, the study of real-life examples is crucial. Throughout the design course, we devote several lectures completely to the discussion of design examples. Typically, these cover more complex implementations that break apart in smaller design problems. Good design examples are not easy to find, but to students, they seem to be the high point of a design course, according to their course evaluation feedback.
- **A frosting of languages:** Languages and modeling techniques make sure you have something to capture design efforts. However, languages should not be the objective of the course. Students may get the impression that a learning a language is the same as learning to design. This is of course deceptive. Learning C is inadequate for learning embedded-system design, and learning Verilog is inadequate for learning digital system design. Therefore, we believe the emphasis of the course should be on the methods, not on the languages.

These three elements provide the building blocks for open-ended design assignments such as the ones discussed above.

VI. Conclusions

This paper summarized our conclusions from teaching a design-oriented class over several years. We identify creativity as an essential element in design, and open-ended design projects as a mechanism to encourage it.

References

[1] E. Guizzo, "The Olin experiment" Spectrum, IEEE , 43(5):30-36, 2006, doi: 10.1109/MSPEC.2006.1628505.

[2] P. Schaumont, "A Practical Introduction to Hardware/Software Codesign - 2nd Edition", Springer 2013, ISBN 978-1-4614-3736-9.

[3] P. Chu, "Embedded SOPC Design with Nios-II Processor and Verilog Examples," Wiley, 2012.

[4] T. Friedman, "Need a Job? Invent It," NY Times, 3/30/2013.