Multi-Processor Programming in the Embedded System Curriculum

Andreas Hansson[1], Benny Akesson[1] and Jef van Meerbergen[1,2]
[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Philips Research Laboratories, Eindhoven, The Netherlands
m.a.hansson@tue.nl

ABSTRACT

Teaching embedded system design is challenging, as the subject covers a wide range of aspects, and also involves skills that students do not learn from a text book. As a result, hands-on projects, with varying degree of complexity, are the most common approach in existing courses. Traditionally, the projects are limited to uni-processor systems, and do not address the complications involved in parallelising applications and mapping them to multi-processor systems.

In this paper, we describe a two-year-old embedded systems design course given at Eindhoven University of Technology. In the course, groups of four students are faced with the problem of putting an embedded JPEG decoder on the market within one semester. The starting point is a decoder written in sequential C and an embedded multi-processor system, running on an FPGA. We describe the ideas and organisation of the course, and give examples of what challenges the students, as well as the instructors, are faced with. We exemplify results and give suggestions to those wishing to teach embedded multi-processor programming elsewhere.

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer and Information Science Education

General Terms: Algorithms, Design, Experimentation

Keywords: Embedded System Design, Education, Multi-Processor System on Chip

1. INTRODUCTION

Embedded systems are characterised by the importance of non-functional requirements, i.e. hard or soft real-time constraints, a limited power budget and limited resources, such as memory footprint [19]. Furthermore, architectures must be programmable to deal with changes in applications [4]. Architectures for embedded systems are the result of a compromise between efficiency and programmability. To limit the design effort a platform-based approach is used, integrating many Intellectual Property (IP) blocks, with multiple processor cores of different types and distributed memories [14]. Examples of those platforms are Nexperia [4] and OMAP [13]. Those are developed in large industrial projects, sometimes involving hundreds of man-years with the bottleneck moving more and more towards software.

The question can be asked how to teach this at university. What should students know and how can they learn it? Are hands-on exercises possible? PC-like systems are covered by good classes and excellent material, but this is not so obvious for embedded platforms. In this paper, we describe the approach followed at Eindhoven University of Technology. In the context of a master program on embedded systems, which is a joint program of electrical engineering and computer science, there is a coherent set of four courses tackling the aforementioned questions. Following a bottom-up approach, the first course focuses on the lower levels of design, i.e. logic and register-transfer level synthesis used to develop IP blocks like ALUs, multipliers, and memories, with a focus on FPGA implementation. The second course uses those IP blocks to build a wide range of processor cores spanning the whole spectrum from fully programmable microprocessors to digital signal processors and application-specific instruction set processors. The third course discusses the communication between those cores and Network on Chips (NoC) play a central role. The fourth course, Embedded Systems Laboratory, is devoted to 5 ECTS credits (140 hours distributed across 15 weeks) of hands-on design exercise, integrating the previous courses and applying the lessons learned. This paper, extending on [12], describes this fourth course. The course concepts presented in this paper have been adopted by Delft Technical University, where a similar course was given by the Computer Engineering department for the first time last year.

The Embedded Systems Laboratory is similar to project-based courses given at other universities [2, 5, 15, 17, 20] in that it integrates skills from a diverse set of subjects, e.g. programming, processor architecture, computer organisation and NoCs. Most students are familiar with these subjects, but few have any experience in integrating the skills [5]. We share the observation that hands-on sessions are indispensable to acquire the necessary skills [2], with a good balance between practical knowledge and fundamental understanding [5]. The course therefore consists of a large project assignment, focusing on the process of mapping a particular application, in this case a JPEG decoder, onto an embedded multi-processor platform. The assignment empha-

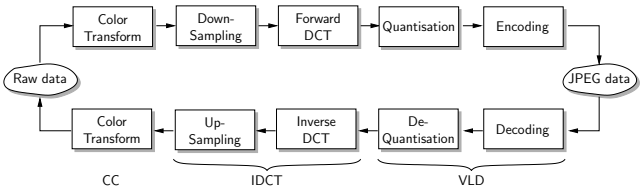

Figure 1: JPEG encoding and decoding.

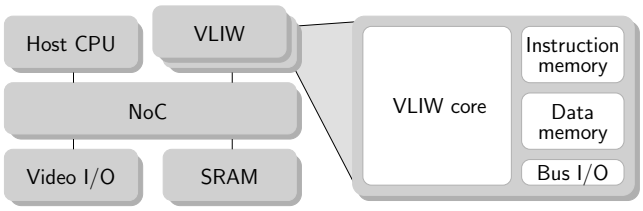

Figure 2: Architecture template.

sises the growing importance of software in embedded systems [15, 17, 24], and resource-limited performance-oriented design [15, 22], but also involves challenges in areas like personal time management and teamwork, similar to [5, 8]. Unlike the aforementioned works, this paper emphasises the challenges involved in going from uni- to multi- processor systems, and the importance of communication and synchronisation. In doing so, it extends on [12] with a more elaborate exposition of the goals, ideas and methodologies underpinning the teaching and organisation of the course. Additionally, this work also discusses challenges with re-examination in group-based projects, and presents course evaluation results.

The remainder of this paper is organised as follows. In Section 2 we discuss the starting point of the assignment, being the given application and hardware platform. Section 3 focuses on the course itself, explaining what the goals are and how the work is organised. We also discuss our interaction with the student groups and give a structured overview of the course. Next, we elaborate the challenges involved in porting and parallelising the application in Section 4, followed by a discussion on performance evaluation in Section 5. We highlight examination and grading in Section 6, before looking at course evaluation in Section 7. Lastly, Section 8 contains a discussion, followed by conclusions in Section 9.

2. ASSIGNMENT STARTING POINT

In this section, we discuss the starting point of the assignment, which is to: *Put an embedded JPEG decoder on the market in 15 weeks.* We start by giving a brief introduction to the concepts of JPEG decoding in Section 2.1. We then proceed by presenting the hardware platform and its building blocks in Section 2.2. Lastly, we discus how the development environment brings the hardware and software together in Section 2.3.

2.1 Application

The application used in the course is a fully functional JPEG decoder written in ANSI C [9]. Decoding a JPEG image is a non-trivial task involving similar steps as many other media codecs, such as MP3, AAC, and H264. The core of all the aforementioned standards is a discrete cosine transform that transforms data into the frequency domain. To achieve a good compression ratio, the transformation into the frequency domain is combined with other techniques, like quantisation and run-length encoding.

The JPEG decoder can be generalised into three main decoding stages, shown in Figure 1: Variable Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT) and Colour Conversion (CC). The VLD decodes the variable-length encoded JPEG data, dequantises it and arranges it into blocks of 8x8 values, referred to as minimum coded units (MCU). The MCUs initially contain frequency data, but are transformed from the frequency domain to the pixel domain, and up-scaled if necessary, by the IDCT step. After this transformation, the blocks contain image data in the YCbCr colour format that the CC step converts to RGB.

The JPEG decoder is suitable for the course for a number of reasons: 1) It offers a reasonable amount of code with which to become familiar (less than 1500 lines of code). Some of the students have never encountered C before, and the JPEG decoder has proven to be manageable also for them. 2) The original decoder makes use of dynamic memory allocation (`malloc`/`free`) and file I/O (`fgetc`/`fputc`), something that is not natively supported by the target embedded platform. 3) The simpler JPEG decoder retains the technical complexities of its audio and video counterparts, thus giving the application good educational value. 4) The decoding is data dependent, in that the different decoding steps require a varying amount of computation (and communication) for different images. 5) JPEG decoding is not trivially parallel. The VLD is inherently sequential, whereas the IDCT and CC are easy to parallelise. This leads to a wide variety of parallelisations, trying to overcome the restrictions imposed by the JPEG format. 6) The decoder is small enough to fit in the local instruction memory of one core. Thereby, it is possible to break the assignment into several steps, where the decoder is first ported, then optimised and parallelised. 7) The decoder has the benefit of being familiar to the students and fun to work with, as the results can be presented on a screen attached to the actual hardware platform. As also observed by Edwards [5], video is visually satisfying when it works, and it can be debugged by inspecting the displayed images. 8) By decoding a sequence of JPEG images, the decoder is turned into a primitive video player, adding soft real-time aspects to the assignment.

2.2 Hardware platform

The platform used in the course, depicted in Figure 2, builds on the concept of using multiple distributed computational and storage resources, interconnected by a scalable NoC. The architecture instance we present the students with consists of three uniform Silicon Hive [23] Very Long Instruction Word (VLIW) cores, a large off-chip SRAM, and a frame buffer for video output. In addition, a general-purpose host CPU is attached to the system. The different components are interconnected by an instance of the Æthereal NoC [7]. A brief discussion on the different building blocks follows.

For the course, we use a simple, three-issue-slot Silicon Hive

VLIW, without a floating-point unit, and with private instruction and data memory, each of 32 kbyte. The memories are also accessible through a slave port on the processors' bus interfaces, forming a distributed memory together with the dedicated memory tile.

In addition to the data memory in each core, the platform instance has only one external memory, as is commonly the case, either for cost reasons or due to a limited number of pins [16]. In addition to the background memory, the cores can also write to a frame buffer, where a designated display controller presents the contents on a DVI output port. This functionality is used during the laboratory sessions to get immediate visual feedback of the results.

The host is a general-purpose processor that is responsible for the initialisation and orchestration of the hardware resources, e.g. loading the binaries to the VLIWs and configuring the NoC and memory arbiters. Although the host processor is typically a part of the SoC, we choose to map the host interface on the NoC to an external PC. This enables the students to use their own PCs, running Linux, as host CPUs during the labs.

All the aforementioned hardware blocks are interconnected by the Æthereal NoC [7]. In Æthereal, the different master and slave interfaces are logically interconnected by *connections*. A connection can be seen as virtual wires, offering a certain throughput and latency guarantee. Together, a set of connections forms a *use-case*, which acts as a virtual on-chip infrastructure. Network resources are pre-allocated for a number of given use-cases, using the UMARS tool [11], and it is left as an exercise for the students to choose an appropriate use-case for their specific JPEG decoder implementations.

A complete architecture instance is mapped to a Xilinx Virtex4 LX-160 FPGA [25], fitted on an Agility RC340 board [1]. The RC340 offers a range of peripherals (audio, video, USB, memories) that are particularly attractive, as it enables results to be shown in far more elaborate ways than blinking LEDs. The board is available in the classroom during the laboratory sessions, connected to a server PC via USB, and the students connect to this PC, in turn, via the local network, similar to what is described in [3]. Through this network connection, they can upload bitfiles to the FPGA, and also have their PCs act as the host CPU, once the device is configured. By allowing network connections to the server with the board, it is also possible to work remotely between sessions, something that is greatly appreciated and extensively used by the students.

The platform is suitable for the course for a number of reasons: 1) The type of programmable multi-processor platform is representative for signal-processing architectures where low power, and support for many features and standards is imperative, 2) The hardware is fixed, but the resource allocation of the NoC is chosen by the students from a set of precomputed use-cases [10], depending on which resources need to communicate and their throughput and latency requirements. 3) The embedded cores have only fixed-point arithmetic, no operating system, no caches and explicit memory management, reflecting the situation encountered in many contemporary SoCs. 4) The students get to experience real hardware, and not only simulation or modelling. This increases the number of issues that the students have to solve, and also the amount of work (and risk) involved in teaching the course. The reward, however, is that the students can see tangible results of their efforts, something that has shown to be a great motivator [5]. 5) Using industrially relevant IP components and tools provide real-world experience and open up opportunities for the companies involved to head hunt students for internships and employment. 6) The platform offers a complete system simulation environment for functional verification, performance evaluation and debugging. This is the topic of the following section.

Table 1: Simulator levels of abstraction.

Level	Instruction scheduling	Computational accuracy	Speed (OP/s)
c-run	no	integer	∼G
unscheduled	no	bit	∼10M
scheduled	yes	bit	∼M
fpga	yes	bit	50M

2.3 Development environment

The Silicon Hive cores are supplied together with a retargetable compiler, assembler and linker, as well as a complete simulation environment. As illustrated in Table 1, the Silicon Hive development environment offers a number of abstraction levels, going from fast checking of the functional correctness, to cycle-accurate simulation. First, in c-run mode, all code is compiled with `gcc`, to verify that the algorithm is working for a supplied set of reference images. Second, in unscheduled mode, the code that is to be run on the cores is compiled with the Silicon Hive compiler, but is not scheduled to instructions slots, registers, etc. This enables the programmer to generate code with instruction semantics of the specified core. Third, in scheduled mode, the compiled code is scheduled to maximise instruction-level parallelism, and the programmer thus gets a complete view of the utilisation of the core's resources, i.e. the register files and intra-core interconnect. In this step, the tools also provide feedback about memory usage, instruction slot scheduling, and detailed profiling information. The fourth and last mode uses the FPGA with the student's computer acting as a host. The host then loads the microcode to the embedded cores on the FPGA and starts the execution.

To reduce the amount of low-level programming, we provide the students with example code (although not optimised) that demonstrate how to start cores, read and write to the external memory, open network connections, etc. As a result, during the first lab session, the students start with a simple example application (adding two numbers), adapt it for the embedded platform, simulate until they achieve the desired behaviour, and run it on the actual FPGA.

The availability of development tools and support libraries removes much tedious and error-prone work, and thus enables the students to focus on the higher-level issues involved in programming multi-processor systems. The one big problem with the proprietary tools is that it complicates the interpretation of any potential error messages. When using

`gcc`, for example, Google can most likely help to decipher compiler and linker errors. That knowledge now has to come from the teachers, who should thus be familiar with the tools and the architecture. In our case, all teaching is done by people that are actively using, extending and researching on the platform. We share the view of [17], that integrated development environments shield the students from the compilation process. Indeed, many students express that they develop a new level of understanding after using `make` and a command-line based cross-compilation environment.

In conclusion, the important benefits of the development environment are: 1) It offers multiple levels of abstraction with seamless transitions between the levels. 2) Command-line tools make all steps involved in compiling and executing an application explicit. 3) Support libraries and example programs enable the students to focus on the higher levels of design. 4) The environment enables the designer to evaluate solutions based on quantitative information about memory usage, instruction scheduling etc.

3. OVERALL STRUCTURE

This section starts by introducing the course goals. There after, we illustrate how we organise and support the students to reach those goals. This involves dividing the students into groups, supplying the necessary infrastructure, and continuously following up and interacting with the students.

3.1 Course goals

We divide the course goals into *low-level* and *high-level* goals. These are closely related to the *core competencies* we identify as desirable, if not necessary, for a graduating student that is to design, program, and use multi-core embedded systems.

Starting with the low-level goals, that are closely related to the *practical experiences* in the course, the students should *know* how to: 1) use the provided development and simulation environment, i.e. to compile, map, simulate and evaluate their programs, 2) port sequential C code to the target VLIW core, i.e. know what limitations the embedded cores impose and how to use the board-level peripherals instead of file and terminal I/O, and 3) parallelise the application, i.e. split a sequential program into concurrently executing tasks to exploit data-level and task-level parallelism. We see those goals directly reflected in the steps taken to solve the assignment.

All these low-level goals are stepping stones, means to reach the high-level goals. The student should *understand*: 1) how embedded and desktop programming differs, 2) how multi- and uni-processor programming differs, 3) the difficulties involved in defining and evaluating the performance of an embedded application, and 4) how design decisions impact the quality of the solution. Moreover, they should show an *ability to*: 1) handle their individual responsibilities within the group, 2) interact with other students within and between groups, and 3) report their conclusions and findings to peers and teachers.

The course goals are reflected in the organisation and structure of the course (this section), in the way the assignment is solved by the students (Section 4.1), and finally in the examination and grading (Section 6).

3.2 Working in groups

From the first laboratory session, the students are divided into groups of four people. Similar to [15], we treat each group like a start-up, and effort is made to ensure that all groups are multi disciplinary and multi cultural, and hence contain students with different educational and cultural backgrounds. The groups assign roles with different responsibilities to their members, much like what is proposed in [15]. The four roles are: application expert, architecture expert, embedded programming expert, and group leader.

The task of the application expert involves learning the details of the JPEG decoding algorithm, and to identify the important functions in the code and their interfaces. The architecture expert focuses on the details of the processing cores, NoC and memories. The embedded programming expert learns how to port and upload code to the embedded VLIW core, and how to use the system support libraries. Lastly, the group leader is responsible for dividing the work among the members, reporting the group's progress, and helping the other members wherever needed.

3.3 Supporting infrastructure

The basic infrastructure offered to the students consists of the server machine with the FPGA board attached and a cluster of Linux computers where the development environment is installed. As already mentioned, these facilities are available for remote connections at all times, not only during lab hours.

Similar to [3], we see that the group-oriented assignment emphasises best-practise software design principles, including revisions control, test-suite validation and group code review. For this purpose, we provide `subversion` and `cvs`, and the facilities for using `cron`. Many groups use this functionality in combination with the remote access, and make it a habit to run nightly test suites and benchmarks. An added benefit of this practise is that is helps reduce the load on the FPGA during the lab sessions.

In addition to the software and hardware tools, the students have access to a Wiki that is used for communication between teachers and students, but also for communication among students, both within and between groups. Much of the content on the Wiki, such as questions and answers about problems relating to tooling or hardware, is reusable between instances of the course, resulting in a continuous refinement of supporting material.

3.4 Interaction with the students

Apart from interacting with the students in the lab and on the Wiki, we have defined a number of additional forums. One of the most important forums are bi-weekly group meetings, where the progress of the individual groups is discussed. These meetings last roughly 20 min during which we give feedback to the group and help with risk management to ensure that they provide their deliverables on time.

After two or three weeks of the course, once all groups have

gotten sufficiently far to be able to contribute, we also arrange meetings for students with specific roles. In these meetings we discuss the difficulties (and potential solutions) that are unique to the FPGA expert, group leader, etc. We believe that organising meetings for individual group members strengthens their identity with the assigned roles, and also tests their ability to communicate and share with the rest of the group. These meetings also allow the groups to help each other with organisational as well as implementation issues, while still maintaining a competitive atmosphere. Also in the classroom, we encourage students to ask their peers (also outside their group) for help before consulting a teacher. Our experience is that this approach works well and that the groups still present unique solutions.

In addition to the aforementioned meetings, we also have a benchmark committee, comprised of representatives from all groups. This committee is summoned on two or three occasions, starting from when the groups have a working single core implementation of the decoder. At this time, most students have learned enough about the application and platform to discuss and determine suitable performance metrics. In the most recent instance of the course, a Wiki page was created where all groups posted their most recent benchmark results. We clearly noticed that this page spurred the competition between the groups and motivated some students to put in additional effort, similar to what is observed in [21].

4. ASSIGNMENT SOLVING

During the first week, the students focus on familiarising with the development environment by mapping educational examples to the platform. This is accompanied by presentations introducing the VLIW cores, the NoC, the FPGA, the support libraries etc. The slides, together with publications on the application and the architectural building blocks, form the lecture material. Much of this material is available at the course web site [6].

At the end of the first week, when the group members have had some time to familiarise with each other, they assign the different roles, and the work on the actual JPEG decoder starts. The work is divided in three distinct phases: 1) porting the application to execute on a single core on the target platform (Section 4.1), 2) parallelising the application to use multiple cores (Section 4.2), and 3) optimising the solutions to improve performance (Section 5). The step-by-step arrangement is important as it makes it easier to organise the work into smaller, yet meaningful parts, with intermediate deliverables and deadlines. This is something we have found necessary to accurately track the progress of the groups.

4.1 Porting the application

The JPEG application is distributed as sequential C code that executes on a normal desktop PC. The first challenge the students are faced with is to port the code to execute on a single VLIW core. The major issues to solve involve memory management, and handling of console and file I/O.

No standard library function is provided for dynamic memory allocation, since the memory architecture is non-uniform and distributed, creating multiple placement options. Memory allocations are hence done statically, and the programmer determines if a particular variable should be mapped to the limited amount of faster local memory of the core, or to the larger but slower external memory. A challenge in this step is that statically allocating arrays requires algorithmic knowledge from the programmer, since they must be dimensioned for the worst case.

The original JPEG decoder uses file system I/O to read the encoded bit stream and to write the decoded image. However, the provided architecture does not have a file system. Instead, the core must read the encoded image from the external memory, which is the only memory large enough to store it, and write the decoded image to the frame buffer. The host is used to transfer the encoded image from the file to the external memory, which requires familiarity with the system support libraries for communication between the host application and the FPGA.

During the porting, the students learn to appreciate the different refinement steps of the development environment. It quickly becomes apparent that the FPGA offers tremendous speed, but complicates debugging due to the lack of observability, thus presenting realistic obstacles to traditional debugging techniques [3]. This is partly a result of how the students access the FPGA, as the network connection makes it difficult, if not impossible, to use facilities like ChipScope [25]. Most groups resort to using the cycle-level simulation environment, and only use the hardware for the final functional verification and benchmarking. Approximately four weeks into the course, the decoder is ported and working on the FPGA. This is when the parallelisation begins.

4.2 Parallelising the application

After successfully porting the application to the target platform and performing initial benchmarks, the groups proceed by parallelising the application to make use of multiple cores. We require each group to implement and benchmark at least two different parallelisations. The two most common solutions involve exploiting *data parallelism*, by allowing multiple cores to work on different parts of the image, and *functional parallelism*, where the decoding functions are mapped to the different cores. Many variations of these solutions have been explored during the course, including hybrid versions that aim to combine the best of both. In this article, we limit the discussion to the two basic solutions, which are presented next.

The idea of a data parallel implementation of the application is that multiple cores are assigned to decode different parts of the image. A benefit of this approach is that very few changes are required to the ported code executing on a single core. This parallelisation is so simple that some groups even manage to go from a single-core decoder to a first data parallel decoder during one lab session of three hours.

The first question that the students have to answer is how to divide the image among the cores. Different strategies distribute the complexity of the image differently among the cores, as illustrated in Figure 3. Another issue with the data-parallel solution is that the VLD is inherently sequential, and it is not possible to know exactly where a block begins without decoding the previous ones. This implies that

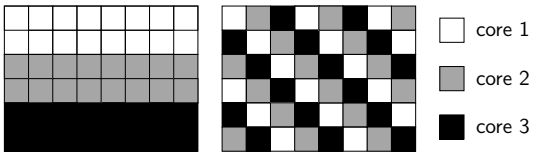

Figure 3: Strategies for distributing MCUs.

all cores must read the encoded image from external memory and perform the VLD, although they skip the IDCT and colour conversion if the current MCU block does not correspond to their assigned part of the image. Reading the encoded image multiple times increases memory contention, affecting performance negatively. This is seen in the results of the initial parallelisation, where the groups report speed-ups of roughly 1.7 and 2.3 times for 2 and 3 cores, respectively. The students thus get to experience the limited scalability of this approach, and explore solutions to mitigate the effect.

A more scalable solution exploits functional parallelism by mapping the decoding functions to different cores. An important challenge is to determine how to partition the functions among the different cores to get a balanced load and to minimise inter-core communication. For this purpose, the students use the cycle-accurate simulation model that gives detailed profiling information. Another difficulty the students are faced with in the functional partitioning is that different pictures place very different computational requirements on the functions in the decoding algorithm. This is illustrated in Figure 4, where the work-load distribution for two different pictures, *Noisy* and *Quiet*, is shown. The *data-dependent behaviour* makes it extremely difficult to partition the decoder in a way that creates a good balance between the cores for all pictures.

The most important learning experience associated with the functional decoder stems from the fact that the students must implement inter-core communication and synchronisation. The groups typically start by synchronising using simple flags. Eventually, most groups go for a double-buffered approach, or decide to use circular buffers with read and write pointers. An implementation of the C-HEAP protocol [18] is also provided with the hardware platform for reference. In implementing the inter-task communication, emphasis is put on hiding the latency of the memory subsystem, i.e. the NoC and memory arbiter and controller. The

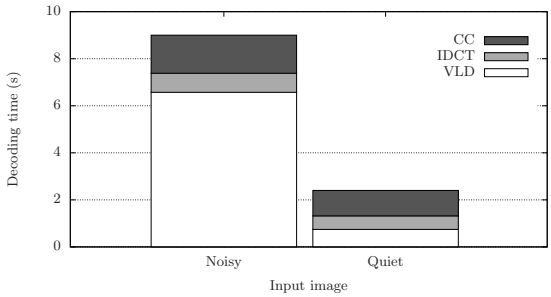

Figure 4: Work-load distribution.

students thus have to employ concepts like posted writes and burst transfers.

5. ADDRESSING PERFORMANCE

Once a solution is functionally correct, an iterative optimisation and benchmarking phase begins to improve its quality. In Section 5.1 we elaborate on the benchmarking procedure. We then discuss optimisations for a decoder executing on a single core in Section 5.2 and on multiple cores in Section 5.3.

5.1 Benchmarks

The groups are encouraged to continuously evaluate their designs using quantitative benchmarks throughout the development process. This allows them to directly see the impact of design decisions on quality, and learn about the trade-offs involved. The benchmarking procedure is standardised by a committee comprised of representatives from all groups, as already discussed in Section 3.4. This ensures that all groups are familiar with the procedure and that their results are comparable. Although a large variety of metrics is discussed, ranging from processor utilisation to design effort, the committee typically decides to use metrics that are relevant to the user or system designer and easy to measure and compare, such as decoding time and memory foot-print.

Decoding time is measured by starting a timer on the host after uploading the encoded JPEG image to the external memory. After starting the timer, the host starts the three cores and waits until all of them have completed. A benefit of this benchmarking method is that it is easy to implement, although a drawback of the approach is that the time required for the host to start the cores and to detect that they finished execution is captured by the measurement. Since the host processor is connected via USB, this overhead may add up to a second to the decoding time. For future instances of the course, we are planning to use an on-chip cycle counter for time measurement. Benchmarking the memory requirements of a solution is simple, as the tooling outputs the required amount of instruction and data memory for every core.

Similarly to what is reported in [21], we have observed that the students use greatly varying techniques to improve their benchmark results. Some groups simply adopt a trial-and-error approach, while other groups systematically identify and address bottlenecks. We also see that most students are aware of optimisations for general processors, but are not familiar with the opportunities that present themselves in a multi-processor system.

5.2 Optimisations for a single-core decoder

The optimisation process is guided by profiling the code using the cycle accurate simulator. Profiling helps identifying functions that are called often or require a lot of time to execute, indicating that they may be good candidates for optimisation.

Optimisations targeting the single core decoder can be categorised as: 1) algorithmic short cuts, 2) adaptations to fit with the computational cores, and 3) adaptations to fit better with the communication infrastructure. The first category involves using knowledge about the JPEG decoding algorithm to speed up decoding, such as throwing away higher

frequency components, or exploiting common cases in the image format. The second category concerns making the computation more efficient by adapting it to the processor core architecture to get a more efficient instruction schedule. The third category considers rewriting the code to reduce the number of memory accesses. For the first category, the programmer must have deep insight into the JPEG algorithm. The latter two categories require the programmer to be intimately familiar with the target architecture and tooling.

The most influential algorithmic short cut in JPEG decoding is that of IDCT-bypassing. That is, when an MCU is unicoloured and does not contain any frequency components, the IDCT can be skipped. The short cut does not compromise the result, and in the case of the *Quiet* benchmark image, more than half of the MCUs are skipped. Another important optimisation is that of detecting common colour encodings in the CC. Most JPEGs use only two types of encoding (4:2:2 and 1:1:1), and by implementing special CC functions for these common cases, the indexing in the CC is greatly simplified.

There are many opportunities to improve the JPEG decoding time by exploiting knowledge of the processor core architecture. One of the major adaptations, done by many groups, is to replace the given Loeffler IDCT with Chen-Wang IDCT. The latter uses fewer multiplications and is better matched to the VLIW in question. By further adapting the code to use variables rather than arrays, the load on the register banks increases, but extra transfers to memory are avoided, resulting in a net gain. Another technique that we have seen examples of in the CC is to use look-up tables with precomputed values.

The last category of optimisations targets the memory architecture, aiming to reduce the number of accesses to remote memories, and to use the accesses more efficiently. A significant speed-up is achieved by using local memory rather than the shared external memory (or the memory of another core). The size is, however, very limited, and not all data will fit in the local memories. To use the remote memory accesses more efficiently, the code must be adapted to read/write whole words rather than sub-words, such as characters. The latter optimisation, for example, reduces the time required for the VLD by almost two times.

5.3 Optimisations for parallel decoders

The optimisations used for the single-core decoder are also applicable to the parallel decoders, but it is quickly noted by the students that the speed-ups observed for the single-core solution are not reflected when they are applied to code that runs on multiple cores. This demonstrates the influence that communication and synchronisation have on the decoding time and serves as an important experience for the students.

There are refinements of data parallel implementations, addressing the memory contention issue. One such refinement involves ensuring that only one core performs the VLD on a particular line and shares the important results with the other cores through a structure in memory, allowing them to skip the line. This optimisation reduces the decoding time for both the aforementioned images by approximately 15%. A drawback of this refinement is that additional memory (approximately 2 kbyte) is required to store the information shared by the cores. The optimisations for the functionally pipelined version mostly considers moving smaller blocks of code between the cores to improve the load balance, or reducing the amount of data that is communicated between the cores. We have also seen numerous examples of different inter-task communication methodologies, aiming to reduce the cost of data transfers.

6. EXAMINATION AND GRADING

Students are *graded individually* on a scale from one to ten, where six and above are passing grades. Similar to [15], grading is based on visible, concrete contributions to the final solutions. The grading process starts with an assessment of the group output in the form of a demonstration of at least two working parallelised solutions, a report of about six pages, and a group presentation for the rest of the class. Each student also gives an oral exam of 20 minutes. Half of this time the students present their individual contributions to the group. This is followed by a Q&A session where we ask the students questions related to their area of responsibility. With our organisation of the exams, it takes two days to grade 30 students. At least two teachers are present at each exam to improve objectivity and fairness. Thanks to the interaction in the classroom, most of the grading can be done before the oral exams, but having both is beneficial for the slightly less extrovert students. The final grades are determined by consensus between all teachers involved.

Dealing efficiently and fairly with failing students is challenging when combining individual examination and grading with group-based project assignments. Developing customised extra assignments to be carried out by individual students is too time consuming to be considered a feasible solution, due to the additional supervision and tool support required. Therefore, our approach to re-examination involves reducing the time spent by the teachers by focusing on the high-level goals. Failing students are provided with the reports of all groups participating in the course and are instructed to consider the benefits and drawbacks of the proposed solutions, and to explain similarities and differences in their conclusions. We encourage the failing students to discuss the reports with the students in the other groups to enhance the learning process. The re-examination is in the form of an oral exam where the student presents the results of the assignment, followed by a discussion. Passing the re-examination results in a minimum passing grade, emphasising that it is not an alternative to the regular assignment and examination.

7. EVALUATION

Students are encouraged to evaluate courses using a web-based system that asks them to rate different aspects of a course on a scale from 1 (bad) to 5 (very good). The system also allows students to provide feedback to the teachers as free text. We received course evaluations from both 2007 and 2008. In both cases, approximately 30% of the participating students filled out the evaluation. Next, we discuss some observations from the evaluation results. The presented numbers are an average from the results of the two instances of the course, unless stated otherwise.

The course scored very well in general with an overall rating of 4.4 both years. The evaluation results also show that the students consider the course very relevant to the educational program (4.7). The usefulness of the course and study material is considered good (4.1), and the students consider the examination/assignment to have high quality (4.3). The students furthermore think that the organisation of the course improved significantly from the first to the second instance (3.5 to 4.2), probably because the course concepts were solidified, and the teaching material was improved. According to the evaluations, the students furthermore consider the course to be slightly on the difficult side and think that they spend more effort on the course than suggested by the indicated study load of 140 hours (3.4, where higher than 3 is too difficult/too much time). All in all, the evaluations suggest that the students enjoyed participating in the course very much (4.5).

Student feedback indicated that the workload quite often turns out to be unevenly distributed within the groups, typically resulting in more work for the group leader and embedded system expert and less for the hardware and application experts. We noted that some students interpret the roles as a rigid structure and stand idle if there are no tasks falling within the responsibilities associated with their role. Many students suggest distributing the four people over two sub-groups, focused on exploring different parallelisations, instead of working individually according to the assigned roles. After roughly half the course, six weeks, most groups resorted to such an organisation. Furthermore, our experience is that groups should not be larger than four persons. During the last instance of the course, we had a few groups of five people. When asked after the course, all but one such group did not consider having five members an advantage.

In addition to the self-assessment provided by the students, our own assessment of student learning suggests that the course succeeds in achieving its goals. All groups this far managed to solve the assignment in the given time frame, which requires extensive work towards the low-level goals. One of the most important indicators that the high-level goals have been reached, in addition to the actual grading, is the amount of graduation projects following up on the issues that are highlighted in the course. Thanks to the course, we see an increased interest in such projects, and the students are perceived to have a good understanding of the issues unique to multi-processor embedded systems.

8. DISCUSSION

Similar to [15], we focus on the embedded software aspects from a systems perspective. The hardware is given. Still, the platform offers a great amount of mapping decisions, with distributed memory, multiple processors, and several NoC use-cases to choose from. More freedom could of course be given to the students, but we do not deem it feasible to do so with the current 140 hours of the course. Moreover, we try to minimise the problems involving understanding and complying with idiosyncratic interfaces and I/O devices. We have found that, in contrast to our first beliefs, any help with these low-level issues saves a lot of valuable time, without compromising the educational value of the course. In fact, we believe that the relevancy of the course is improved by moving the focus from particular APIs and specification formats to higher-level issues.

The students appreciate the problem-based nature and report that they specifically learn about the concepts behind JPEG compression, different parallelisations, and the methodology and importance of exploring different architectural mappings. During the course, the student groups evaluate roughly four or five different partitionings, and get to experience the non-linear speed-up. In their presentations they often conclude with the observation that adding more hardware is not the solution. More important than one more core is to optimise the use of the ones that are already there.

As in [15], we observe that students retain knowledge better when working through actual implementations that force them to confront the very real limitations and quirks of embedded systems. That said, we feel it is important to reduce the amount of practical knowledge and tool fighting as much as possible. From the first to the second year we also simplified a lot of the lower-level issues, and now help the students to get by the first hurdles in using the system. The amount of tutorial exercises and demonstrative examples is also significantly larger.

We share the observation in [15] that guest speakers from embedded system industry helps motivating the students. For this reason, we invite representatives from the companies providing our hardware platform with IP components to present themselves, the technology they provide, and the significance of the course in their context.

As a response to student evaluations, in future instances of the course, we will recommend the groups to assign responsibilities to the group members instead of assigning roles and derive responsibilities from there. We will furthermore encourage the students to dynamically redistribute responsibilities, if required, to balance the workload.

We believe that the Embedded System Laboratory delivers a level of realism that helps in both motivating the students, and reinforcing the experiences gained during the course. Based on student evaluations, the course succeeds in bringing together knowledge from other classes and teaches the students skills that they do not learn from a text book [5], e.g. the balance between top-down and bottom-up design, the ability to seek and find the information you need, the ability to debug and reason about observed behaviour, and to understand the different factors affecting the performance of a multi-processor system.

We continue to review and extend the course. With the maturity of the existing tooling and IP we have the opportunity to add more elements to the course and offer even more freedom, e.g. customisation of the processors and NoC instance. This will, however, be optional, as we already find the course sufficiently challenging.

9. CONCLUSIONS

In the Embedded Systems Laboratory, the students become familiar with many of the difficulties involved in programming multi-processor embedded systems. By the end of the course, they have successfully ported a JPEG decoder to the target multi-processor platform and evaluated a range

of parallelisations on an actual FPGA instance. The assignment presents many challenges, ranging from working in a group to choosing the right compiler directives for a critical piece of an algorithm.

Embedded Systems Laboratory ran for the second time in 2008 with 31 participating master students, both from the Electrical Engineering and International Masters programme on Embedded Systems. The course concepts have furthermore been adopted by Delft Technical University, where a similar course was given by the Computer Engineering department for the first time this year. Next year, we aim to keep improving the course in line with feedback from students and teachers, and prepare another class of students for the problems we are facing with the wide-spread adoption of multi-processor embedded systems.

10. REFERENCES

[1] Agility DS. http://www.agilityds.com, 2008.

[2] P. Bertels *et al.* Gathering skills for embedded systems design. In *Proc. WESE*, 2007.

[3] D. Brylow and B. Ramamurthy. Nexos: A next generation embedded systems laboratory. In *Proc. WESE*, 2008.

[4] S. Dutta *et al.* Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*, 2001.

[5] S. Edwards. Experiences teaching an FPGA-based embedded systems class. *ACM SIGBED Review*, 2(4), 2005.

[6] Embedded systems laboratory. http://www.es.ele.tue.nl/education/EmbeddedSystems.html, 2008.

[7] K. Goossens *et al.* The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. and Test of Comp.*, 2005.

[8] M. Grimheden and M. Törngren. How should embedded systems be taught?: Experiences and snapshots from swedish higher engineering education. *ACM SIGBED Review*, 2(4), 2005.

[9] P. Guerrier. *Un Réseau D'Interconnexion pour Systémes Intégrés*. PhD thesis, Université Paris VI, 2000.

[10] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *Proc. NOCS*, 2007.

[11] A. Hansson *et al.* Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. DATE*, 2007.

[12] A. Hansson *et al.* Multi-processor programming in the embedded system curriculum. In *Proc. WESE*, 2008.

[13] J. Helmig. Developing core software technologies for TI's OMAP platform. *Texas Instruments*, 2002.

[14] K. Keutzer *et al.* System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(12), 2000.

[15] P. Koopman *et al.* Undergraduate embedded system education at carnegie mellon. *ACM TECS*, 4(3), 2005.

[16] J. Leijten *et al.* Prophid: a platform-based design method. *Des. Autom. for Emb. Syst.*, 6(1), 2000.

[17] J. Muppala. Bringing embedded software closer to computer science students. *ACM SIGBED Review*, 4(1), 2007.

[18] A. Nieuwland *et al.* C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. *Des. Autom. for Emb. Syst.*, 7(3), 2002.

[19] C. Rowen and S. Leibson. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. Prentice Hall PTR, 2004.

[20] A. Sangiovanni-Vincentelli and A. Pinto. Embedded system education: a new paradigm for engineering schools? *ACM SIGBED Review*, 2(4), 2005.

[21] P. Schaumont. Hardware/software co-design is a starting point in embedded systems architecture education. In *Proc. WESE*, 2008.

[22] C. Shih *et al.* Toward HW/SW integration: A networked embedded system course in Taiwan. *ACM SIGBED Review*, 4(1), 2007.

[23] Silicon Hive. http://www.siliconhive.com, 2008.

[24] S. Tsao, T. Huang, and C. King. The development and deployment of embedded software curricula in Taiwan. *ACM SIGBED Review*, 4(1), 2007.

[25] Xilinx, Inc. http://www.xilinx.com, 2008.