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, the students, divided in groups of four, 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 are faced with. We exemplify results and give suggestions to instructors wishing to teach embedded multi-processor programming elsewhere.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*

General Terms

Design, Algorithms, 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 [17]. Furthermore, architectures must be programmable to deal with changes in applications [3]. 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 [12]. Examples of those platforms are Nexperia [3] and OMAP [10]. 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 supported by good classes (mainly in the CS dept) 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 EE and CS, 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, memories etc. The focus is 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 (120 hours distributed across 12 weeks) of hands-on design exercise, integrating the previous courses and applying the lessons learnt in those courses. This paper describes this fourth course.

The Embedded Systems Laboratory is similar to project-based courses given at other universities [2, 4, 13, 15, 18] in that it integrates skills from a diverse set of subjects, e.g. programming, processor architecture, computer organisation and NoCs. Most students have experience in these subjects, but few have any experience integrating the skills [4]. 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 [4]. The course therefore consists of one large project, focusing on the process of mapping a particular behaviour, in this case a JPEG decoder, onto an embedded

multi-processor platform. The assignment emphasises the growing importance of software in embedded systems [13, 15], and resource-limited performance-oriented design [13, 19], but also involves challenges in areas like personal time management and teamwork, similar to [4]. In contrast to the aforementioned works, the emphasis is on the challenges involved in going from uni- to multi- processor systems, and the importance of communication and synchronisation.

The remainder of this paper is organised as follows. In Section 2 we discuss the starting point of the assignment, being the given application, the hardware platform, and the associated tooling and development environment. Section 3 focuses on the assignment itself, explaining what the students have to do and how they are organised. We also discus our interaction with the student groups and give a structured overview of the course. Next, we elaborate on what challenges are involved in porting the application to the hardware platform in Section 4 and discuss the parallelisation in Section 5. Section 6 highlights the performance evaluation, followed by a discussion in Section 7. Finally, conclusions are presented in Section 8.

2. ASSIGNMENT STARTING POINT

In this section, we discuss the starting point of the assignment, which is to map a JPEG decoder onto a multi-processor platform. 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. We end this section with an overview of the development environment in Section 2.3.

2.1 Application

The application used in the course is a fully functional JPEG decoder written in ANSI C [7]. 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.

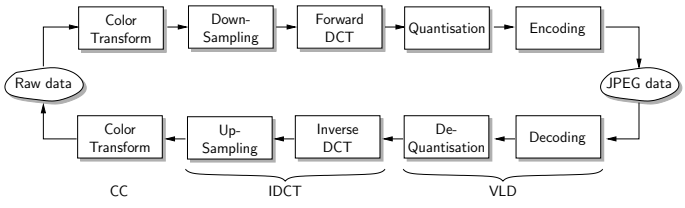

Figure 1: JPEG encoding and decoding.

The JPEG decoder is suitable for the course for a number of reasons: 1) It offers a reasonable amount of code to familiarise with. Some of the students have never encountered C before, and the JPEG decoder proves to be manageable also for those students. 2) The original decoder makes use of dynamic memory allocation and file I/O, 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 processor. Thereby, it is possible to break the assignment into several steps, where the decoder is first ported, then optimised and parallelised. Solving one problem at a time also gives intermediate deliverables and deadlines, something we have found necessary in tracking the progress. 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 [4], video is visually satisfying when it works, and it can be debugged by inspecting the displayed images. 8) By decoding sequential JPEG images, the decoder is turned into a primitive video player, adding real-time aspects to the assignment, without causing faulty behaviour if the requirements are not satisfied. Again, this helps in splitting the assignment into manageable parts.

2.2 Hardware platform

The platform used in the course builds on the concept of using multiple distributed computational and storage resources, interconnected by a scalable NoC. Using this type of programmable embedded platform in the course is representative for signal-processing architectures where low power, and support for many features and standards is imperative.

Like Edwards [4], we want the students to experience real hardware, and not only simulation or modelling. This is made possible thanks to Silicon Hive [20], providing their low-cost, low-power domain-specific Very Large Instruction Word (VLIW) processor cores, and NXP, providing the Æthereal NoC interconnect fabric [6]. The students hence work with industrially relevant IP components and tools. Using an actual hardware instance of the platform increases the amount of issues that the students have to solve, with more tools to manage and more things that can go wrong. It also increases 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.

The architecture we present the students with, depicted in Figure 2, consists of three uniform Silicon Hive cores, a large off-chip SRAM, and a frame buffer for video output. In

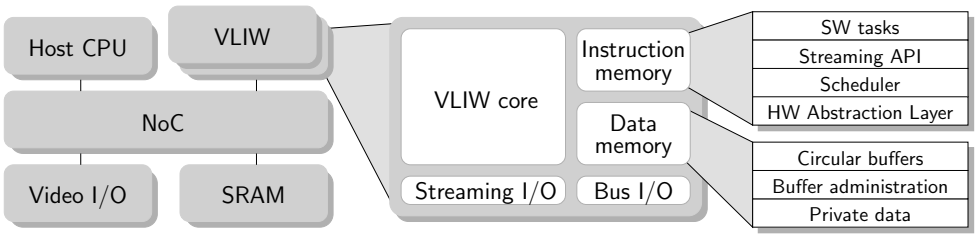

Figure 2: Architecture template.

addition, a general-purpose host CPU is attached to the system. The different components are interconnected by an instance of the Æthereal NoC [6]. A brief discussion on the different building blocks follow.

The Silicon Hive VLIW cores are customisable, making it possible to adapt the costs and the performances of the various computation nodes to a given application. For the course, we use a simple, three-issue-slot architecture, without a floating-point unit. The cores use a memory-mapped architecture and have a master interface to enable reads and writes to memories external to the processor. As shown in Figure 2, every processor also has its own private instruction and data memory. The memories are also accessible through a slave port on the processors bus interfaces, forming a distributed memory together with the dedicated memory tiles. The challenges that arise due to the selected processor architecture are: 1) the application must use fixed-point arithmetic, 2) multiplication and load/store operations compete for the same instruction slot, 3) the application must fit in a local memory of 32 kbyte (which is chosen to just barely fit the complete JPEG decoder), 4) the processor core has no caches and requires explicit memory management, 5) the core has no operating system, thus leaving any task scheduling to the programmer.

In addition to the 32 kbyte of data memory in each core, a central memory tile, here after referred to as *external memory*, provides 8 Mbyte of SRAM. While being significantly larger, the external memory has an access time an order of magnitude larger than the local memories. This is due to the traversal of the NoC, and the sharing of the memory read and write port. 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 [14]. 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. Note that in contrast to [4], the students do not have to implement any low-level drivers to interface with e.g. DVI and USB interfaces. We believe that earlier courses are sufficient in teaching the lower levels of design, and leaving these elements out gives more time to the higher-level issues we want to emphasize.

As seen in Figure 2, the system contains not only VLIW processing cores, but also a host CPU. Like in the IBM Cell [11], 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 connections [8] 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 ports of the aforementioned hardware blocks are physically interconnected by the Æthereal NoC [6]. 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 [9], and it is left as an exercise for the students to choose an appropriate use-case for their specific JPEG decoder implementations. The hardware is thus fixed, but the programming of the NoC is chosen from a set of pre-computed use-cases [8]. The students select a use-case based on which master and slave ports that need to be interconnected for their specific parallelisation, and what throughput and latency the desire between the different connections.

A complete architecture instance is mapped to a Xilinx Virtex4 LX-160 FPGA [21], fitted on an Agility RC340 board [1]. In contrast to [4] that uses many inexpensive boards, we are using a single board, which sells for about $10.000. Since we are targeting multi-processor systems, a fairly large FPGA device is needed (our current system uses more than 50.000 LUTs). Moreover, 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. Interfacing with the peripherals is also fairly straight forward thanks to APIs from Agility.

The FPGA 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. 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. By these means, having a single board for 30 students has proven to be manageable, although more boards would of course be advantageous. Only during the last week of the course, when all student groups are preparing for the evaluation, did we experience that students lost considerable time just waiting to access the board.

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. In the Silicon Hive development environment, a number of steps are possible, going from fast checking of the functional correctness, to cycle-accurate simulation. First, all code is compiled with `gcc`, to verify that the algorithm is working for a supplied set of reference images. Second, the code that is to be run on the cores is compiled with `hivecc`, but not scheduled to instructions slots, registers, etc. This enables the programmer to generate code with instruction semantics of the specified core. Third, the compiled code is scheduled to maximise Instruction Level Parallelism (ILP), 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 step 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.

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 instructors. Teaching assistants 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 [15], 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.

To even further reduce the amount of low-level programming, we provide the students with functional example programs (although not optimised) that demonstrate how to read and write to the background memory, closely mimicking the behaviour of `fgetc`, `fseek` and `ftell`. They also get example code that shows how to interface with the frame buffer and the display controller. As a result, already 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. We have found that, in contrast to our first beliefs, any help on this stage 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 idiosyncratic APIs and specification formats to higher-level issues.

3. ASSIGNMENT OVERVIEW

In this section we describe how the assignment, to parallelise and map the JPEG application on the presented hardware platform, is actually carried out, from an organisational point of view.

3.1 Course structure

Already from the first laboratory session, the students are divided into groups of four people and presented with a problem: *Put an embedded JPEG decoder on the market in three months.* Similar to [13], we treat each team 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.

During the first week, the students focus on familiarising with the development environment by mapping educational examples to the platform. This is accompanied by lectures introducing the VLIW cores, the NoC, the FPGA, the support libraries etc. The slides, together with a wide range of publications on the architectural building blocks, form the lecture material. Moreover, the students have access to a Wiki, also containing useful information from past year's courses. All of this is available at the course web site [5].

After an initial week of introductory exercises of tutorial nature, the teams assign roles with different responsibilities to their members, much like what is proposed in [13]. The four roles are: 1) application expert, 2) architecture expert, 3) embedded programming expert, and 4) 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 team progress, and helping the team wherever needed.

Experience shows that the workload quite often turns out to be unevenly distributed. Many students suggest to distribute the four people over two sub-groups, focused on exploring different parallelisations. This also gives the benefit of always doing pair programming. After roughly half the course, six weeks, most groups resorted to such an organisation, and in future instances we will recommend such an approach. 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 reported that it would even have been advantageous to have four members instead of five.

Once the roles are assigned, 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, 2) parallelising the application to use multiple cores, and 3) optimising the solutions to improve performance. The step-by-step arrangement is important as it makes it easier for the students to organise their work into smaller, yet meaningful parts. It also simplifies setting partial deadlines that we follow up on by means of a group page on the course Wiki. Updating this page is one of the most important responsibilities of the group leader. In addition to the Wiki, we also discus the progress of the individual groups during bi-weekly meetings, where each group gets roughly 20-30 minutes time with the teaching assistants.

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 discus the difficulties (and any potential solutions) that are unique to the FPGA expert, group leader, etc. This allows the groups to help each other with organisational as well as implementation issues, while still maintaining a competitive atmosphere between the groups. Also in the classroom, we encourage students to ask their peers (also outside their group) for help before consulting a teaching assistant. Our experience is that this approach works well and that the groups still present unique solutions.

3.2 Examination

To pass the course, each team has to present: a demonstration of at least two working solutions, a design document of about six pages, and a group presentation (and demonstration) for the rest of the class. Since students are graded individually on a scale from one to ten, where six and above is a passing grade, each student also gives an individual presentation and present their individual contributions during an oral exam. Thanks to the interaction in the classroom, most of the grading can be done before the individual presentations, but having both is beneficial for the slightly less extrovert students. Moreover, the individual presentations are also an excellent opportunity to get the students' opinions on other fellow students. Often, the students are remarkably honest and not afraid to share their opinions. Similar to [13], grading is based on visible, concrete contributions to the final solutions (based on what role the student had in the group). They are evaluated as application experts, group leaders etc.

4. PORTING THE APPLICATION

The JPEG application is distributed as sequential C code that executes on a normal desktop PC. The first challenge of the design teams 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, 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 application, in its original form, makes rich use of the console to print debug information in case there is something wrong in the implementation or the encoded image. The target embedded system has no means of outputting this information, since it does not have a console. Printing this information is, however, very useful to limit debugging time in case errors are introduced in the code during the porting effort. For this reason, these commands are not removed from the code, but rather redefined to empty statements by the pre-processor before compilation for the VLIW core. This allows all debug information to still be printed if the code is compiled for a regular computer to verify its functional correctness.

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. The decoded image is also written back to external memory during development, allowing the host application to read the output and compare to a reference image that was decoded before porting the code. Automating this procedure allows bugs introduced during porting to be discovered quickly.

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. 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 [21]. Most groups resort to using the cycle-level simulation environment, and only use the hardware for the final functional verification. Approximately four weeks into the course, the decoder is ported and working on the FPGA. This is when the parallelisation begins.

5. PARALLELISING THE APPLICATION

After successfully porting the application to the target platform and performing initial benchmarks, the design teams proceed by parallelising the application to make use of multiple cores. As mentioned in Section 3, the assessment criteria 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 in Sections 5.1 and 5.2, respectively.

5.1 Data parallelism

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. All cores execute the same program, but use a unique identifier to determine which part of the image to decode. 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. This is illustrated in Figure 3 where the image is par-

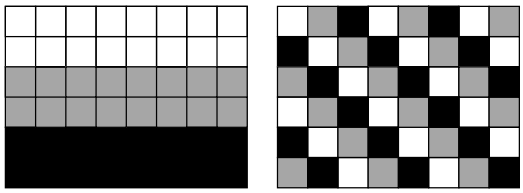

Figure 3: Strategies for data partitioning

titioned among three cores according to the different shades of grey. Dividing the image in three horizontal slices, as done in the left part of the figure, would create an unbalanced load in a scenic picture with a blue sky in the top, since significantly less computation is required by the IDCT for this part of the image. Another strategy that is better in this respect is tiling, shown in the right part of the figure, where a core decodes every third MCU block.

The main drawback with the data parallel JPEG decoder 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 all cores must read the encoded image from external memory and perform the VLD, although the IDCT and colour conversion is skipped if the current MCU block does not correspond to its assigned part of the image. This is also 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. Consider, for example, the partitioning strategy to the left in Figure 3, which only requires the first core to read 1/3 of the image from external memory, and the second core 2/3, while the last core must read all of it. This can be compared to the tiling strategy on the right in the figure, which requires all cores to read the entire memory, increasing memory contention. Several groups also develop synchronisation schemes that allow the cores to share information about the parts of the image that are already decoded.

5.2 Functional parallelism

In this solution, the decoding functions are mapped to the different cores, creating a pipeline where an MCU block is processed by all the cores in sequence before decoding is complete and it is written to the frame buffer. 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, giving detailed profiling information. A common way to split the decoder is according to the three stages, VLD, IDCT, and CC that were explained in Section 2.1. This partitioning has the benefit of providing clear interfaces between the functions where only frequency blocks and pixel blocks are communicated between the cores.

A 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. Figure 4 shows the decoding time required for the VLD, IDCT, and CC, respectively, on a single core. The two

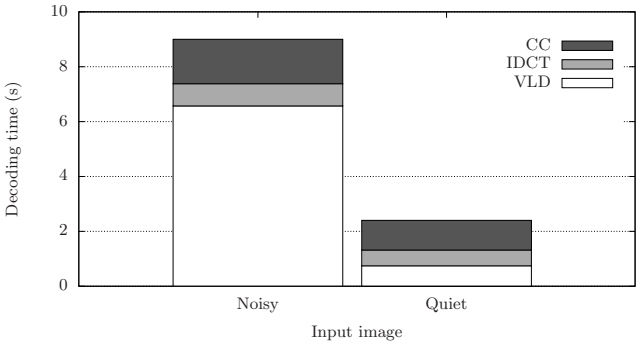

Figure 4: Single-core profiling.

images are both XGA resolution (1024 x 768), but *Noise* (748 kbytes) contains a lot of high frequency information and is dominated by the VLD, whereas *Quiet* (53 kbytes) contains mostly uni-coloured MCUs and is fairly balanced. This shows that 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.

In addition to the difficulties in splitting the decoder into three equal parts, the students must also 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 [16] is also provided with the hardware platform for reference. The students also evaluate the effects of placement of buffer data and administration, as well as the size of the inter-task buffers. Profiling is also carried out by looking at how often the various tasks stall on full or empty buffers. 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 students thus have to employ concepts like posted writes and burst transfers.

6. ADDRESSING PERFORMANCE

The two times we have given the course, all the student groups have completed the project, meaning that they have at least two parallel JPEG decoders working by the end of the course. To get higher grades, we also require that the students assess the performance of their decoders. Thus, the evaluation is not only functional, but involves performance measures such as execution time, time-to-market and memory footprint.

Once a solution is functionally correct, an iterative optimisation and benchmarking phase begins to improve its quality. In Section 6.1 we elaborate on the benchmarking procedure. We then discuss optimisations for a decoder executing on a single core in Section 6.2 and on multiple cores in Section 6.3.

6.1 Benchmarks

The teams are encouraged to continuously evaluate their designs through 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 design teams. This ensures that all teams are familiar with the procedure, and that their results are comparable. The standardised benchmarks consider two aspects of embedded systems being performance, in this case decoding time, and memory requirements.

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 on on-chip cycle counter for time measurement. Benchmarking the memory requirements of a solution is simple, as the required amount of instruction and data memory is output by the tooling for every core.

We have observed that the students use greatly varying techniques to improve their benchmark results. Some groups simply adopt a trial-and-error approach. Other groups have given example of systematically identifying and addressing 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 multiprocessor system.

6.2 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.

6.3 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 has on the decoding time.

There are refinements of data parallel implementations, addressing the memory contention. 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.

7. DISCUSSION

Similar to [13], 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 5 ECTS credits of the course. Moreover, we try to minimise the problems involving understanding and complying with idiosyncratic interfaces and I/O devices.

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 presentation 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 there.

As in [13], 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 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 [4], 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.

8. CONCLUSIONS

In the Embedded Systems Laboratory, the students get to familiarise 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 the 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 teaching assistants, and prepare another class of students for the problems we are facing with the wide-spread adoption of multi-processor embedded systems.

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

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

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

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

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

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

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

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

[11] J. Kahle *et al.* Introduction to the Cell multiprocessor. *IBM Jour. of Research and Develop.*, 49(4/5), 2005.

[12] 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.

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

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

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

[16] 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.

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

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

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

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

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