A PROCESSOR-COPROCESSOR ARCHITECTURE FOR HIGH END VIDEO APPLICATIONS

Elmar Maas, Dirk Herrmann, Rolf Ernst, Peter Rüffer

Technische Universität Braunschweig, Germany
maas@ida.ing.tu-bs.de

Sieghard Hasenzahl, Martin Seitz

Philips BTS, ICC Weiterstadt, Germany
100726.1616@compuserve.com

ABSTRACT

High end video applications are still implemented in hardware consisting of many components. Integration of these components on one IC is difficult as they are typically low volume products and often customization is also required, e.g. in studio applications. This is easier on the board level than on an integrated system. Using hardware parameters for customization can partly overcome the flexibility problem with additional hardware costs. Low cost can be obtained by a change in the architecture paradigm to a processor-coprocessor system. This, however, requires careful design space exploration since the performance target is beyond current DSP processors while at the same time flexibility is required. The paper presents the application of high level synthesis [1] and novel Hardware-Software Co-Synthesis tools to design space exploration. It is shown that completely different algorithms can be mapped to the same target system at much a lower cost than the current approaches.

1. INTRODUCTION

As of now, *real-time* computing of high end studio video applications requires dedicated hardware. Developing dedicated hardware is a time consuming and thus an expensive task. Moreover a long development time increases the time to market. Once the hardware platform is implemented, it is difficult to make changes, but industrial experience in this low volume market shows that hardware adaptation to customer needs is required in almost every design. Thus, flexibility of the hardware platform is a major design issue. With our design approach we are targeting at both, reducing development time and cost, and improving hardware flexibility. This design approach was employed in an industrial cooperation project. In order to work on relevant examples, we chose two algorithms, each representing one of the two extremes in video signal processing, i.e. demand for computing power and memory bandwidth requirement. Starting with a specification of these algorithms in C-language, we used our Hardware-Software Co-Design environment COSYMA [2] for detailed design space exploration. In order to be able to use the COSYMA system, which was originally developed for designing embedded systems, we defined an architecture template (figure 1) consisting of a processor and a coprocessor which are synchronized with the environment by buffer memories. It is necessary to buffer the fixed rate video I/O to allow for non-trivial processor-coprocessor design solutions. COSYMA, along with its high level synthesis (HLS) system BSS[1] [3], allowed us not only to do useful code optimization but also to efficiently prune the design space to find an

[1] Braunschweig Synthesis System

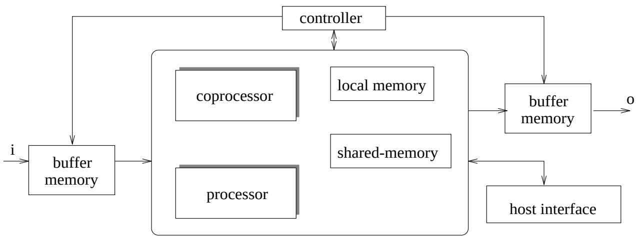

Figure 1: System Architecture Template

optimal architecture for the given signal processing tasks. In the following sections we describe

- our design flow, which is non-standard in the area of signal processing,
- the requirements that had to be met and
- the design space exploration and final system integration.

2. DEVELOPMENT ENVIRONMENT

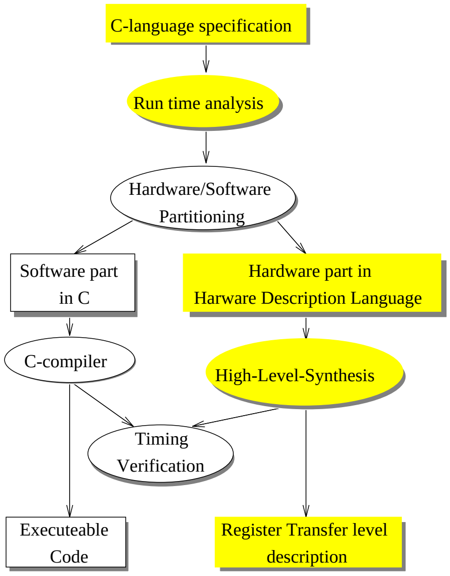

Figure 2: The COSYMA Approach to Hardware-Software Co-Design

Figure 2 shows the COSYMA approach to Hardware-Software Co-Design: A given C-specification with additional timing constraints is analyzed and automatically partitioned into software and hardware parts. For the software part, a C-program is generated which can be handled by standard compilers. The hardware part is translated into a hardware description language on behavioural level which is further processed by our HLS-system. BSS generates an RT-level description which is synthesizeable by commercial systems like the Synopsys Design-Compiler.

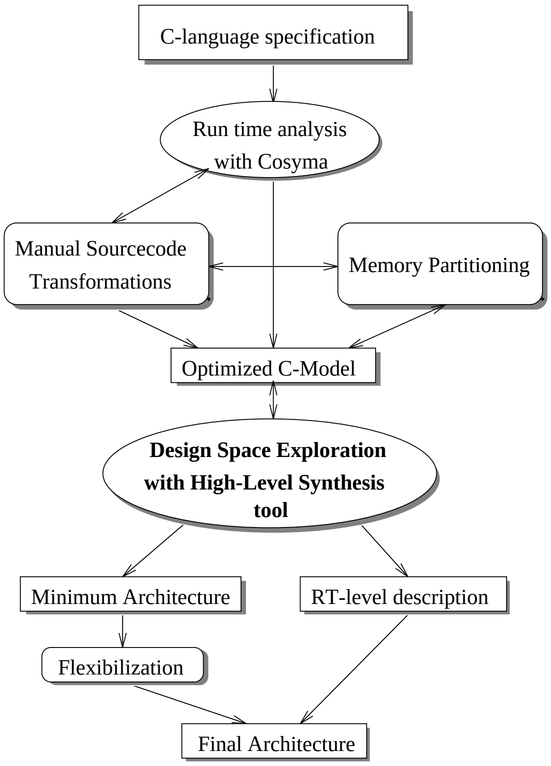

Figure 3: Design Flow in the Project

In this project we mainly used the tools from the hardware path highlighted in fig. 2 and shown in more detail in fig. 3. A signal processing algorithm in C, which was developed by the industrial partner, was evaluated with COSYMA. The requirements concerning run time and memory bandwidth, which will be discussed in detail in the next section, made manual source code transformations inevitable. These transformations were driven by the following goals:

- reducing the number of operations,
- reducing the number of memory accesses and
- increasing potential parallelism.

We executed loop merging for pairs of consecutive loops which performed computations on every pixel of the input image and thus ran over the same index range. This saved the evaluation of loop conditions, and since the results of the first loop could directly be used as an input for the corresponding iteration of the second loop, the memory traffic was reduced. As a side effect, potential parallelism for HLS was increased. This transformation is known from parallel compilers [4] [5]. Some functions, given as abstract behavioural descriptions, were rewritten to save operations: for example, a function for computing the median of five values, originally implemented by bubble-sorting, was replaced by a manually optimized solution which (in this function) reduced the number of comparisons by 56% and swap operations by 30%. Some source code transformations changed the number of memory accesses so that an existing memory partition had to be adapted. After changing the memory partition, restructuring of the source code often was required or considered advantageous. The effects of these transformations on system performance and cost were evaluated by the output of COSYMA's run time analysis as well as from the scheduler of its HLS tool. The optimization steps were iterated until the timing constraints were met and the number of external memories was minimized. The outcome of this iterative process was an optimized C-model, an RT-level description for the coprocessor and a minimum hardware architecture on board level. The latter was manually extended in order to allow special computation modes and to make the board reusable for other purposes.[2]

3. EXAMINING THE SOFTWARE

3.1. Some observations

We selected two applications which are highly relevant in practice: One is a chroma key (blue screen) algorithm which requires an extremely large number of computations and the other is a median noise reducer filter which is characterized by high memory bandwidth. In the beginning, there was little knowledge of the computational power and memory bandwidth requirements of these algorithms, as prior to this project, the complete knowledge of the applications was hidden in a large number of individually developed hardware modules. Therefore, the first step was to extract this knowledge and put it into a software specification. The result was a software description of the operations on the hardware platform which was by no means optimal for a DSP software implementation. It represents one design point and the aim was to prune the design space. An efficient small processor-coprocessor implementation required manual algorithmic transformations. The possible number of transformations is very large and the effect of these transformations on memory bandwidth and performance is not always easy to anticipate. Thus, design space exploration starting with a hardware solution is cumbersome.

To get a first impression, we started with a simplified algorithm to find a potentially useful set of transformations, a first memory architecture and a coarse estimation of the required performance. This first version of the chroma key algorithm was named qad (quick and dirty) since it was still functionally incomplete but from the viewpoint of the software designer it represented a coarse measure for the final version. This algorithm was gradually completed while the design architect started with the design space exploration. Figure 4 shows how the requirements of our two algorithms have changed over a few months. The data on which the diagrams are based has been created by COSYMA's RT-simulator for an LSI SPARC processor as a reference under the assumption that all data and programs were in first level cache. Compilation was done using the GNU gcc from the Free Software Foundation. Starting with the first version of the qad in December 1994, we could reduce the number of operations by almost one third simply by performing code transformations such as loop merging,

[2] It may be noted that reusability represents one aspect of flexibility.

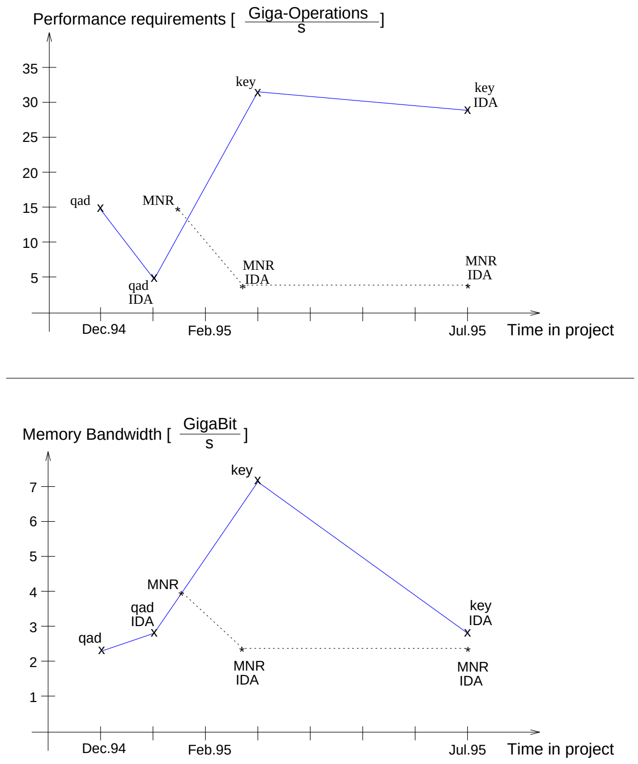

Figure 4: Requirements vs. Project Runtime

partial unrolling etc. The optimizations were done with respect to a processor-coprocessor system and a HLS system in mind, i.e. the original structure of the code was completely destroyed. Both the memory bandwidth and the performance requirements drastically increased when the algorithm was completed in the next quarter, the latter to more than 30 Giga-Operations/s (SPARC-instructions). One reason for this increase was the introduction of parameters for different standards[3] or to take care of potential customer requirements. The required computing effort was not anticipated which indicates that it can be difficult to estimate the performance of an algorithm implemented earlier on a completely different architecture. However, it is important to know the required computing effort to choose an appropriate computing platform.

3.2. Considering a DSP Design

In parallel to our Hardware-Software Co-Design approach we evaluated a software approach. In order to provide as much flexibility as possible, a solution with a programmable DSP would have been favorable at first glance. We decided to take one of the fastest DSPs on the market, the TMS320C80 (MVP) from Texas Instruments [6] [7]. With its four independent DSPs, a master RISC processor and 25 fast 2kByte memory banks on chip, it provides a peak performance of 2 GOPs per second. Unfortunately, the compiler support for the processor is rather poor so that C-code for this processor has to be rewritten in assembly language manually if it has to make use of the performance of the processor. In addition the memory traffic also has to be scheduled by hand Nevertheless, we mapped the median noise reducer algorithm to the MVP, which was smaller and easier to parallelize than the chroma key algorithm. Re-implementing the 166 lines of code for the MVP took almost one man month without taking the training period into consideration. Under the assumption of single cycle RAM accesses, data transfers were performed using the full bandwidth of the MVPs 64-bit bus. However, for the noise filter, the resulting optimized code was still 1.35 times slower than required for real-time, assuming a 50 MHz system. It might have been possible to achieve real-time requirements using two MVPs but this would have required repartitioning of the assembly code. In general, what-if analyses for an MVP system with different parameterization of the code as well as different hardware constellations prove to be very time consuming. In contrast, synthesis, which will be described in the next section, was considerably faster and allowed more efficient design space exploration.

[3]e.g. image size, background colour etc.

4. DESIGNING WITH HLS

Our architecture template (figure 1) consists of a processor and coprocessor. COSYMA's partitioning goal is to place big parts of the algorithm onto the software side, which is executed by the processor, and only smaller and especially timing critical parts on the coprocessor. Assuming the processor is an LSI-SPARC, a speedup of 600(chroma key) or 300(median noise reducer) would be required by partitioning parts to the coprocessor. Taking a faster processor, particularly the MVP, could reduce the required speedups but communication and code generation cannot be done automatically with sufficient efficiency for such a kind of processor. Apart from that we also figured out from the synthesis results that moving relatively small parts of the algorithms to software would not significantly reduce hardware costs for the coprocessor but would increase the memory costs. Eventually the largest part of both algorithms was implemented in the coprocessor.

As a next step we used high level synthesis not only to optimize the mapping of the algorithms to the application specific coprocessor but also to do the design space exploration concerning the board level architecture. Thus, we had two parallel design goals: minimizing the number of computing units (i.e. ALUs, multipliers, etc.) on the chip and finding an optimal number of external memory ports. This optimization problem has been solved iteratively by evaluating the scheduling output of BSS. For example a certain number of ALUs is given to BSS and these ALUs turn out to be utilized concurrently in large parts of the schedule but on the other hand memory ports are idle, the number of ALUs was increased until the ALUs were no longer the bottleneck. Performing memory port optimization required a some more creativity: if the schedule revealed that the memory accesses were the bottleneck, we checked whether the arrays which were accessed could be partitioned to different memory banks.[4] If this was possible the introduction of a new memory port solved the problem. Iteratively applying algorithmic transformations and changing the parameters for HLS we were able to find an optimal board level architecture which suited both the algorithms and had an optimal number of computing units for the chip.

[4]Typically, this involved reformulation of the algorithm with loop splitting while loop merging was used for bandwidth reduction.

Category	mnr	key
Run-time [Mcycles]	1.66	1.87
Number of ALUs	9	30
Number of MULs	2	7
Input data stream [MByte/s]	19.77	39.56
Output data stream [MByte/s]	19.77	29.67
Potential I/O-Bandwidth [MByte/s]	416	349
Required on board memory [MByte]	7	6.25

Table 1: Final Results for the Coprocessor

The final results for the coprocessor are shown in table 1. The algorithms work on full PAL television images (720x576 pixels), so the coprocessor must consume less than 2 million clock cycles per image at 50 MHz clock rate if real-time computation has to be achieved:

$$\frac{40\text{ms}}{\text{image}} * \frac{50\text{ M cycles}}{s} = 2\text{ M cycles}$$

The table shows the minimum numbers of ALUs and multipliers, which BSS instantiated in the coprocessor, for real-time computation of the algorithms. Although the chroma key algorithm is computationally more intensive than the median noise reducer, the latter needs 20% more IO-bandwidth for computation. It is also quite interesting to observe that the algorithms need between 500% and 1000% more IO-bandwidth for computing purposes than for getting data on and off the board. The sum of the memory bandwidths for each algorithm in Table 1 is slightly above the values presented in figure 4 because the table shows the physically provided bandwidths whereas the figure is based on bandwidths obtained as a result of simulation. This reveals that the memory ports of our design are almost used to capacity.

5. OPTIMIZED BOARD ARCHITECTURE

As discussed in the last section, the major parts of the algorithms are implemented on the coprocessor, so we were rather free in the processor selection. Finally, we took the Texas Instruments MVP which potentially allows to perform simple algorithms in real-time. This permits the usage of the prototype board and the MVP programming knowledge for other applications – a matter of knowledge reuse. Apart from the main algorithmic features, the board was extended to allow special computation modes: a frame grab mode permits grabbing a frame out of the processed video stream and passing it on to the host processor via a PCI interface. The input buffer memory was implemented as a double buffer in order to allow MVP non real-time computation without input overflow. These extensions increased board flexibility but they almost doubled the number of pins and wires needed to connect the processors to the memories. Since the coprocessor would have to be connected to all memories anyway, the control logic and the switches were integrated onto it, thus, slightly increasing coprocessor cost but saving hundreds of pins and wires.

The final architecture is shown in figure 5. The prototype-board has a size of 185 mm x 311 mm. The total amount of memory on the prototype board sums up to 7 MBytes SRAM plus three 256k x 8 bit FIFO memories. On a commercial board, the SRAMs would be replaced by highly integrated fast SDRAMs which would reduce both board size and power consumption.

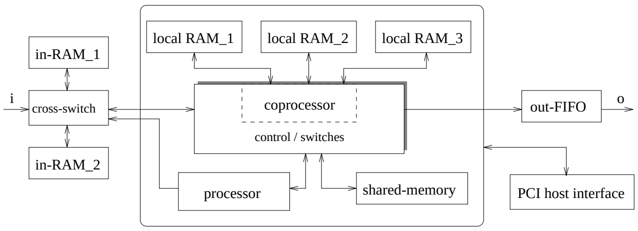

Figure 5: Final Board Architecture

6. CONCLUSIONS

We have shown that by using high level synthesis in the context of Hardware-Software Co-Design for fast design space exploration, a well suited hardware solution for several different applications can be found in a comparatively short period of time. Thus, development costs and time to market could be significantly reduced in future.

An important aspect was the role of flexibility. We found that the flexibility to be able to do "minor" (from the viewpoint of the software developer) changes within the specification could easily double system requirements (section 3.1). With respect to parameterization of the algorithms, which can also be regarded as flexibility , the studying of a competitive DSP solution (section 3.2) revealed that the software approach is not by definition flexible, as well as hardware is not completely inflexible. The implementation of the prototype brought up a third flavour of flexibility, that is the reuse of a design for other than the original purposes. All these manifestations of flexibility had certain impact on the design and eventually increased costs. Flexibility is a relevant factor in a system optimization process and will become more important with an increasing level of integration. It may be noted that designing for flexibility requires a more precise or even formal definition of what it actually is.

7. REFERENCES

[1] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin. *High-Level Synthesis - Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[2] R. Ernst, J. Henkel, and Th. Benner. Hardware–software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[3] U. Holtmann and R. Ernst. Combining mbp-speculative computation and loop pipelining in high-level synthesis. In *ED&TC'95*, pages 550–555, 1995.

[4] U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.

[5] D. E. Hudak and S. G. Abraham. *Compiling Parallel Loops for High Performance Computers*. Kluwer Academic Publishers, 1993.

[6] Texas Instruments, Inc. *TMS320C80(MVP) Online Documentation Reference Set*, 1994.

[7] Texas Instruments, Inc. *TMS320C8x System-Level Synopsys.*, 1995.