A Codesign Case Study in Computer Graphics

Brage, Jens P.; Madsen, Jan

Published in:
Proceedings of the Third International Workshop on Hardware/Software Codesign

Link to article, DOI:
10.1109/HSC.1994.336714

Publication date:
1994

Document Version
Publisher's PDF, also known as Version of record

Citation (APA):
Brage, J. P., & Madsen, J. (1994). A Codesign Case Study in Computer Graphics. In Proceedings of the Third International Workshop on Hardware/Software Codesign (pp. 132-139). IEEE. DOI: 10.1109/HSC.1994.336714

A Codesign Case Study in Computer Graphics

Jens P. Brage Jan Madsen

Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby, Denmark
e-mail: {brage, jan}@id.dtu.dk

Abstract

This paper describes a codesign case study where a computer graphics application is examined with the intention to speed up its execution. The application is specified as a C program, and is characterized by the lack of a simple compute-intensive kernel. The hardware/software partitioning is based on information obtained from software profiling and the resulting design is validated through co-simulation. A locally developed interface model, Merlin, is used as the basis for co-simulation. The achieved speed-up is estimated based on an analysis of profile information.

1 Introduction

Codesign, i.e., the combined development of hardware and software, can be roughly classified as follows:

- Co-development of both hardware and software from a specification which does not favor either implementation strategy.
- Hardware design of instruction set processors. Aside from hardware design, it also involves software analysis to optimize the instruction set.
- Speed-up of an existing software application, by porting parts of the program to hardware.

This paper describes a case study in the latter category: The optimization of an existing computer graphics application written in C. In such applications it is often possible to locate a relatively simple computational kernel, which can then be ported to hardware [4]. This case study reveals a somewhat more complex situation, as the computational load is fairly evenly distributed throughout the program.

In order to analyze the computational load distribution of a program, profiling tools are needed for applications of a realistic size, i.e., several thousands lines of code. Traditional software profiling tools focus on the distribution of CPU time. In a codesign environment, it is equally important to be able to analyze the flow of data between different parts of an application: In a hardware/software environment this will be reflected by physical communication.

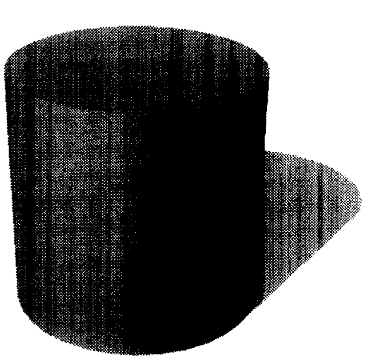

Figure 1: A sample Topoc image; the cylinder is generated by the CSG intersection operator on 8 rotated cubes.

These analysis results are used for hardware/software partitioning. In order to assure that the resulting, partitioned design is still functionally correct, either verification or validation tools are required. These tools must of course be able to handle the semantic differences between hardware and software representations.

Validation of the functional correctness of the resulting design is, in this case study, done by co-simulation. As the input specification is an executable program, it is advantageous to maintain an executable design description throughout the design flow. Thus, the availability of a co-simulation environment is a critical factor in the design methodology.

0-8186-6315-4/94 $04.00 © 1994 IEEE

There are two different methodologies for co-simulation:

- Execution of the software on a simulation of the target CPU.
- Execution of the software in native code on the simulation CPU.

Ecker [3] describes a co-simulation environment based on the first methodology, using VHDL to describe both application specific hardware and the target CPU. This results in a homogeneous environment, but incurs the performance problem of simulating the execution of the software rather than having it run in native code on the simulation CPU.

Others suggest the use of the second methodology. Dun and Jadoul [2] uses TCP/IP to link a Verilog hardware simulator to a natively executing C program. Gibson and Östman [5] uses remote procedure calls to interface a VHDL simulator with software written in C++; the interface is built on C/C++ routines using STYX, a C interface to VHDL. The STYX interface is also used by Hersén [7]; in this approach, co-simulation is a master-slave simulation in which the software acts as the slave.

In this paper, a locally developed interface model for hardware/software systems is used. This model, called Merlin, provides for transaction-based communication between a set of equivalent processes, each of which may represent hardware or software. Thus, this approach is based on the second methodology.

2 The Design Task

Computer graphics represents an important class of applications for codesign, as these applications are characterized by high computation loads and complex algorithms. Each algorithm in a computer graphics application may be classified into one of three main categories:

- Modeling, i.e., the construction of a mathematical representation of some physical objects, the world.
- Rendering. These algorithms convert the mathematical world model into a geometrical scene description.
- Scan conversion, which generates the final bit-map image from the geometrical description.

Modeling and rendering are characterized by highly complex algorithms and medium data rates, whereas scanline conversion is typically simpler but requires much higher data rates.

The application considered in this paper, Topoc, contains all three aspects. Topoc builds a 3D world using objects described as polyhedra and provides a CSG (Constructive Solid Geometry [8]) module to allow complex objects to be constructed from simpler polyhedra. A scene is then constructed from the world model by operations such as hidden-surface removal and shadow casting from point and parallel light sources. This scene is then scan converted into the final image. During scan conversion shading is applied to the surfaces, including surface smoothing and transparency. Figure 1 shows a sample image produced by Topoc.

Generating the sample image in figure 1, which is a *very* simple image, takes a few minutes on a workstation (SUN SPARC station IPX). Thus, the current implementation is far too slow, compared to desirable rendering speeds. The design task described in this paper is the performance optimization of Topoc, by moving parts of the application to dedicated hardware.

Performance improvement may also be obtained by selecting better algorithms, by restructuring the code and the data structures, or even by changing the target CPU. However, in this paper we will not consider these alternatives: The C program is considered the fixed specification for Topoc.

Topoc contains about 8000 lines of C code. A study of the program reveals that Topoc has no simple compute-intensive algorithm kernel, which would form the natural basis of a hardware engine. Thus, speed-up may only be obtained by a detailed study of the application.

3 The Design Process

This section outlines the design process and the following sections then describe each step in detail.

First a detailed analysis is made, to reveal computational bottlenecks; this is used to guide the hardware/software partitioning task. The analysis takes two forms; manual examination of the data and control structures of the program, and automatic profiling by running the application on sample input data. The profiler extracts information on the amount of time spent in each function in the program, and provides an analysis of the structure of function calls, the call-tree. The call-tree and the number of calls of a function are then used to evaluate its relative importance and the amount of data transfers between functions.

Before the final partition is decided upon, architectural considerations must be taken into account. These considerations include the type of coprocessor interface and the memory configurations for the dedicated hardware. For instance, the hardware could be driven by the target CPU or might be running concurrently with its own instruction stream. The memory configuration might be based on shared access to the data store of the target CPU or dedicated memory for the hardware unit.

Once the final partitioned design is decided upon, it must be functionally validated. This is done by separating the design into 'hardware' and software parts, i.e., by forming two new C programs from the original program. As these C programs represent concurrently executing units, the total system can no longer be expressed within the semantics of C; to overcome this problem, the Merlin interface model is initially used to allow concurrent execution.

The next step is to transform the C code 'hardware' description into a real hardware description language, in this case VHDL. Merlin now allows co-simulation of the hardware/software system. Thus, Merlin is used to validate the design during the refinement procedure of the hardware description from initial architectural design in VHDL to final implementation.

The final step is to examine the performance improvement by analyzing the architectural design.

4 Analysis

The manual examination reveals that Topoc uses a relatively small set of data structures to represent geometric data, but that these data structures are used in most computations. The dominant characteristic of the program is a large set of highly complicated expressions based on simple vector operations.

In order to determine the computational distribution of these vector operations, automatic profiling is employed. The main objective of this is to attempt to find localized, computationally expensive kernels, suitable for hardware implementation. This analysis is based on a cylinder object obtained by rotating and merging a cube (see figure 1);[1] this object is complex enough to achieve realistic information, yet simple enough to reduce the overall execution time during profiling.

The profiler provides information on the amount of time (in terms of execution on the simulation CPU, not the target CPU) spent in each function and the distribution of the time among its parent functions. The simulation CPU in the case study is a 40MHz SPARC 2 processor.

At first glance, the main contribution to the execution time of Topoc is the file output function, which spends 48.5% of the total execution time. However, this contribution is irrelevant for the final design, as this will use a true-color frame buffer as the output medium.

In the following, the profiling information is related to the three major computer graphics tasks described in section 2, giving the percentages of the total execution time for each task, after correction for the output functions.

4.1 Modeling

The CSG operations accounts for 13.9% of the total execution time. A careful study of the code reveals that the CSG module is very complex and that no computational kernels above the level of basic vector operations (e.g., vector addition and cross product) can be identified.

4.2 Rendering

The shadow generation and hidden surface removal functions only consumes 0.4% of the total execution time. For the cylinder example this is not surprising, since no shadows are generated and relatively few surfaces are hidden. With more complex examples these functions must be expected to have greater influence. However, the manual examination of the program shows that these algorithms are specialized versions of the CSG functions; so their effect on the partitioning decision can be expected to be an emphasis of the effect from the modeling functions.

4.3 Scan Conversion

Scan conversion accounts for 85.0% of the total execution time. This is distributed on 11.8% to translate polyhedra into 2-dimensional strips and 88.2% to do the actual scan conversion. Thus, the scan conversion may be a subject for further consideration. An investigation of the code reveals that 52.2% of the time spent in scan conversion is spent on basic vector operations.

4.4 Discussion

From the analysis it is evident that the vector arithmetic accounts for a fair amount of the total execution time, in total 34%. As the analysis also shows that there are no algorithmic kernels above this level, the decision is now made to achieve the desired speed-up through the design of a 3D vector arithmetic unit.

In analogy to Amdahl's law [6, p. 586], the total speed-up of the application can be written as:

$$s_t(s_v) = \frac{s_v}{1 + (s_v - 1)r}$$

where s_v is the speed-up of the vector arithmetic achieved by moving it to hardware and $r = 1 - 0.34$ is the fraction of the execution not affected by the application specific hardware. Accordingly, the maximum achievable total speed-up with a vector arithmetic unit is 1.52. Even though this is a fairly limited speed-up, the development will be continued

[1] Unlike figure 1, the analyzed scene does not contain any shadows.

based on this architecture, as the main emphasis is on the design process and not the design itself.

It should be noted that these numbers assume that the target CPU is the same as the simulation CPU. If the target CPU is a less powerful CPU, for instance an MC68EC040,[2] a considerable additional speed-up could be achieved by utilizing the FPUs in the vector arithmetic unit for scalar operations as well. With the available profiling tools, however, it is not possible to analyze this potential speed-up.

5 Architectural Choices

The next step is to choose a suitable architecture for the 3D vector arithmetic unit. The choice is based on an examination of the communication between the application specific hardware and the target CPU. In the following, different options for supplying the hardware with data and instructions are examined.

5.1 Instruction Streams

There are three different ways to supply the instruction stream to the dedicated hardware:

- Common instruction stream (figure 2a); as the CPU and the hardware share the same instruction stream, they are inherently synchronized. The main disadvantage of this model is the reduced instruction bandwidth available to the target CPU. Also, it should be noted that this model requires a CPU which supports coprocessor extensions in its instruction set.

- Command driven (figure 2b); in this case the CPU feeds instructions to the hardware, increasing the load on the CPU. Synchronization is typically achieved by using interrupts as completion signals.

- Multiple instruction streams (figure 2c); this allows the CPU and the hardware to carry out their tasks fully independently. Synchronization is obtained through data exchange. This model avoids the performance problems of the two former models, but typically requires more hardware.

5.2 Data Streams

The data streams for the hardware may be obtained from one of two sources:

- Shared access to CPU memory (figure 3a).

- Local memory for the coprocessor (figure 3b).

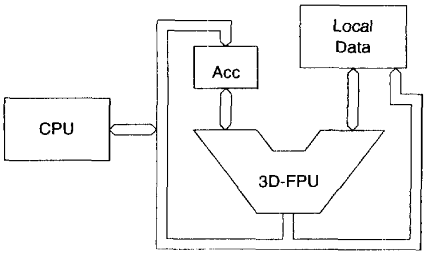

Figure 4: Architecture of the coprocessor; data may be a full 3D vector or a single element.

The advantages of the first approach is that the hardware has direct access to the data structures of the CPU. On the other hand, the second approach avoids bus contention between the CPU and the hardware.

5.3 Choosing the Interface

The results from the analysis are used to choose between the options outlined above.

Considering the lack of any dominating, localized computational kernel in the application, invocation of the dedicated hardware will occur at comparatively high rates. Consequently, the architecture must be chosen for efficient invocation of instructions. On the other hand, as the executed instructions are simple (primitive vector operations), there is little need for complex instruction sequences or explicit synchronization. This leads to a choice of the common instruction stream approach.

From the manual examination of Topoc it turns out that all scalar and vector floating point arithmetic can be allocated on the dedicated hardware; consequently by choosing the local memory model, contention between the CPU and the hardware can be minimized.

5.4 The Internals of the Coprocessor

Now that the interface has been chosen, the final step in the high-level architectural design is to choose the register model for the coprocessor and its instruction set. These choices are based on the decision to place all arithmetic operations on the coprocessor, and on the results of the profiler.

The coprocessor design is a load/store architecture with a single accumulator register as shown in figure 4. All two operand instructions take their input from the accumulator and the local store. The access to the local store can be a full 3D vector at a time or a single element can be picked for scalar operations.

In addition to the local memory, it is also possible to access the data memory of the CPU.

[2] The embedded control version of the M68040, without FPU or MMU.

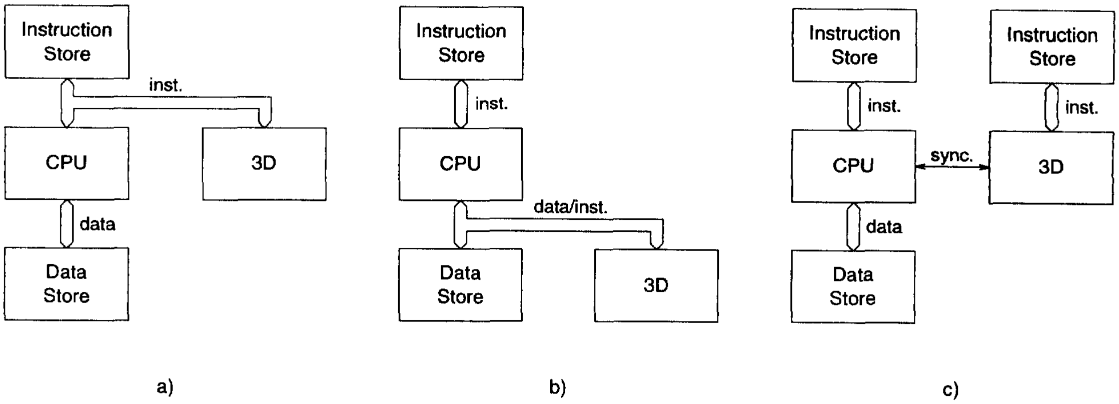

Figure 2: Possible configurations for supplying instructions to the dedicated hardware (3D); a) common instruction stream; b) command driven; c) multiple instruction streams.

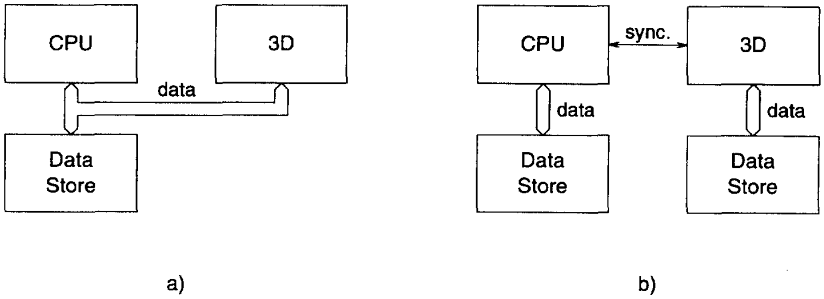

Figure 3: Possible configurations for the data streams; a) shared access; b) local memory.

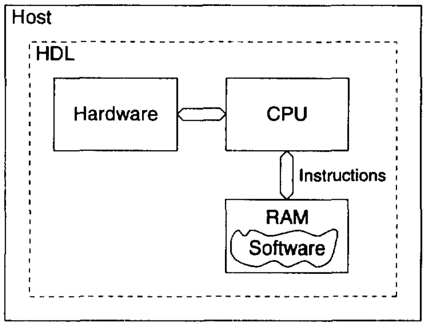

Figure 5: Running the software part on a simulated CPU.

6 Modeling the Resulting Design

In order to ensure that the proposed coprocessor operates correctly in the given application, the entire resulting system, hardware and software, must be simulated.

When a hardware/software system needs to be simulated, there are two approaches:

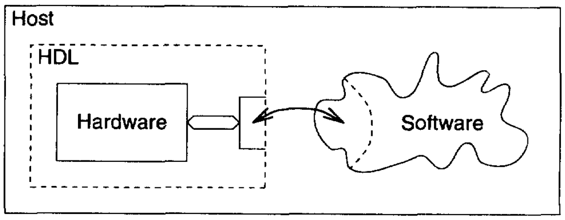

Figure 6: Host native execution of the software part.

1. Modeling the target CPU in a hardware description language, and executing the software on this simulated CPU along with the hardware, see figure 5.
2. Modeling the hardware in a hardware description language, but executing the software directly on the simulation CPU (figure 6).

The first approach has some major advantages: Even if the target CPU is a custom design and either is still in development or simply does not support a hardware simulation environment, it is still possible to obtain accurate simulation results. Also, as everything is modeled within a single hardware description language, this approach avoids the semantic problems of crossing between hardware and software representations.

The second approach seems much less interesting, as it, for instance, does not offer as accurate simulation capability as the first one. However, this may not be a problem, for several reasons: First, in order to make real use of the accuracy of the first approach, a precise model of the computing system is required; effects such as cache misses will have a major performance impact on most modern CPU systems. Modeling such a CPU system with a sufficient degree of fidelity requires a highly detailed model, which is expensive to develop and execute. Also, in many cases a specific CPU has not been chosen in the early design stages; it is only known that a certain block of code is suited for software implementation, and perhaps the designer has an idea about which class of CPU to use (e.g., workstation, large microprocessor or a small embedded core). In this case, the accuracy of a simulated CPU is, of course, useless. Finally, running software on a simulated CPU tends to be extremely time consuming, compared to native execution of a program. As codesign systems often contain quite complex software parts, this may well be prohibitive.

This does leave one major problem with the second approach, though: How to deal with the semantic differences between hardware and software representations, i.e., the differences between code executing natively on a CPU, and code (i.e., the hardware part) being simulated in a hardware simulator (on the same CPU). An easy way to deal with this problem is to define a common interface model for both environments, and then define the total semantics in terms of the events on the interface.

6.1 The Merlin Interface Model

In the case study described here, the second approach is chosen, and the interfaces are described in terms of the Merlin interface model. The Merlin model is an attempt to design a unified interface model for codesign: Rather than designing separate models for different types of hardware interfaces (e.g., bus-based or shared variable) and software interfaces (e.g., DMA/interrupt based or RPC), Merlin aims to provide a single, simple model on which the various interface abstractions can be built.

The Merlin model views a design as consisting of a number of processes; each process may represent either hardware or software, and the processes may communicate by means of three transactions (see figure 7):

- The Attention transaction signals a process of a control event in the originating process.
- The Read transaction allows a process to read a word of data from another process.
- The Write transaction transfers a data word from the originating process to another.

The exact formulation of this transaction-based interface depends on the language used to describe a process, in particular on the pragmatics of the language: For an imperative (software) language like C, the interface is formulated as a set of functions; for a concurrent hardware language like VHDL, the interface consists of a set of signals which implement an asynchronous communication protocol. These signals connect the modeled hardware to a component instance which represent the rest of the system.

The Merlin interface model is not, however, intended to be used directly as the interface model in a given design. Rather, an abstraction layer corresponding to the particular interface intended for a given design should be placed around the Merlin interface. For instance, for a codesign development system, a library of common interface types (e.g., bus-based interfaces) and specific instantiations might be constructed (e.g., VME and ISA busses might be represented).

6.2 Modeling the Coprocessor

In the present case study, a high-level description of the coprocessor interface is desired, as described in the previous section. In order to model this, primitive operations in the original C program, which corresponds to instructions in the coprocessor, are replaced by invocations of Merlin; thus, functions corresponding to coprocessor instructions are used as the interface abstraction.

The coprocessor itself is initially modeled as another process, written in C. This permits the partitioning of the design to be examined and allows validation of the rewrites of the application code. This description is then rewritten in VHDL, in order to more closely represent the chosen architecture.

A prototype simulation environment running under the Unix operating system has been constructed for the Merlin interface model. This system allows a number of software processes (running as native code) and a number of hardware processes (simulated using the commercial VHDL simulator Synopsys) to be run concurrently. Using this system, functional validation of the proposed coprocessor design is carried out.

As the final target CPU and system has not yet been chosen (and thus neither has the low-level design of the coprocessor), it is not possible to give reliable figures for the achieved speed-up. It is, however, possible to estimate

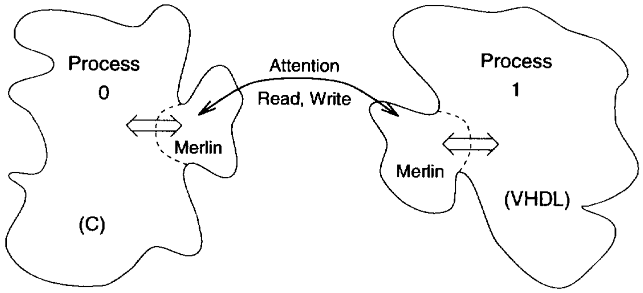

Figure 7: The Merlin interface model: All communication is based on the three transaction types Attention, Read and Write.

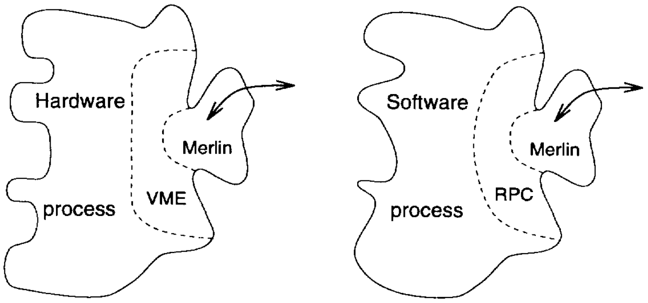

Figure 8: Encapsulating the Merlin interface, here by a VME bus layer and an Remote Procedure Call layer.

the speed-up based on the profiling information and the coprocessor architecture, as described in the next section.

The Merlin implementation has the option of logging all transactions in the system; coupled with the profiling information[3] and given information about the timing of the target CPU and the low-level coprocessor design, this would allow accurate performance figures to be obtained. At the moment, though, the necessary tools to calculate this timing information have not been developed.

7 Performance Evaluation

The profiling information gathered is examined in order to estimate the hardware speed-up factor for the vector functions, s_v. As the target CPU has not been selected as yet, the estimation is based on the assumption that the target CPU is the same as the simulation CPU; this should be a pessimistic assumption, as the target CPU is expected to be less powerful.

If t_v is the total time spent in the vector functions of the original program and t'_v is the speed with the hardware coprocessor, then:

$$s_v = \frac{t_v}{t'_v} = \frac{\sum_{i \in \text{func}} t_i}{\sum_{i \in \text{func}} n_i \cdot c_i / f}$$

where func is the set of vector functions which are now implemented as calls to the hardware. t_i is the time spent in vector function i by the original program and n_i is the number of invocations of the function. c_i denotes the number of cycles used by the hardware, operating at frequency f, in order to execute the vector function i.

Each vector function is implemented as a set of coprocessor instructions. These instructions can be classified according to estimated cycle count; an estimate of the cycle count of each instruction class, o_j, is obtained by studying the M68040 FPUs instruction timings. c_i can now be estimated by examining the implementation of each vector function, counting the number of instructions in each class, $m_{i,j}$:

$$c_i = \sum_{j \in \text{class}} m_{i,j} \cdot o_j$$

Using the results obtained from the profile, the hardware speed-up factor is:

$$s_v = \frac{24.9s}{85.6 \cdot 10^6 / f} = 0.29 \cdot 10^{-6} s \cdot f$$

[3] The timing information from the profile does not reflect the target CPU, but the execution statistics are reliable.

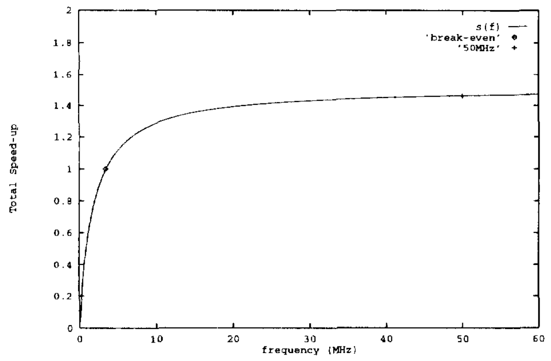

Figure 9: Total speed-up as a function of coprocessor operating frequency. Break-even is achieved at 3.4MHz ($s_v = 1$).

Given s_v, Amdahl's law can be used to obtain the total speed-up as a function of the operating frequency of the coprocessor, as illustrated in figure 9. At a typical operating frequency of 50MHz, a speed-up of 1.46 may be obtained; it should be noted, though, that as in some instruction sequences the coprocessor requires instructions in consecutive cycles, the possible operating frequency is limited by the instruction fetch rate of the target CPU.

8 Conclusions

This paper presents a codesign development task in computer graphics. The goal is to speed up the execution of an existing software application by moving parts to hardware. The application is particularly interesting as it is not possible to locate a simple, compute-intensive algorithmic kernel.

The development is carried out as a codesign case study; consequently, existing design and analysis tools are used wherever possible.

During the design analysis phase, a traditional software profiling tool is used to extract execution information about the source program. From the profiling information, it is possible to locate the computationally most intensive parts of the program. However, the traditional software analysis tools do have significant short-comings in a codesign environment, as they do not provide information on the data flow between algorithmic parts. In the case study, this is resolved by combining the profiling information with manual examination of data transfers between blocks of code.

In order to validate the design after hardware/software partitioning, a co-simulation environment is developed around the Merlin interface model. Merlin is a simplified model which allows a number of processes to communicate; these processes may belong to different semantic domains, i.e., hardware or software. The communication primitives of the Merlin model have been selected to facilitate the construction of a library of physical hardware/software interface types.

As the main goal of this design is to study the design process, only a rudimentary treatment is given to the design itself. For instance, the profiler is only run on a single test case; for a more realistic design, more complex examples should be investigated. Also, it should be noted that the estimated achieved speed-up is fairly low; for a realistic design other approaches, such as pure software optimization, should be investigated.

9 Acknowledgments

Thanks should go to Carsten Christensen for his work on this case study [1]. This research has been sponsored by the Danish Technical Research Council.

References

[1] Carsten Christensen. Coprocessor design from software implementation. Master's thesis, Department of Computer Science, Technical University of Denmark, February 1994.

[2] Johan Van Dun and Luc Jadoul. Hds/H Cosim: a cosimulation prototype applied in the formal design of telecom PBA's. In *Second IFIP International Workshop on Hardware/Software Codesign, Codes/CASHE'93*, May 1993.

[3] W. Ecker. HW/SW co-specification using VHDL. In *Second IFIP International Workshop on Hardware/Software Codesign, Codes/CASHE'93*, May 1993.

[4] R. Ernst, J. Henkel, and T. Benner. Hardware-software codesign of embedded controllers based on hardware-extraction. In *International Workshop on Hardware-Software Codesign, Estes Park, Colorado*, 1992.

[5] Per Gibson and Frederik Östman. Early integration in industrial practice. In *Second IFIP International Workshop on Hardware/Software Codesign, Codes/CASHE'93*, May 1993.

[6] John P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1988.

[7] Rudolf Hersén. Charon - a co-simulation application. In *Second IFIP International Workshop on Hardware/Software Codesign, Codes/CASHE'93*, May 1993.

[8] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes. Constructive solid geometry for polyhedral objects. In *Computer Graphics*. ACM SIGGRAPH, August 1986.