COSYMA: A Software-Oriented Approach to Hardware/Software Codesign

Jörg Henkel
Thomas Benner
Rolf Ernst
Wei Ye
Nikola Serafimov
Gernot Glawe
Technische Universität Braunschweig

We approach hardware/software codesign mainly as a hardware/software partitioning problem. Our software-oriented approach starts with an all-software solution and iteratively extracts hardware components until all constraints are met. The partitioning process is automated to cope with complex real-time systems. We will show that the simulated annealing optimization algorithm offers the opportunity to achieve good results in acceptable computation times. Furthermore, we will outline the translation from software to a hardware description. Finally, full automatically partitioning results will be shown.

1 INTRODUCTION

The area of application for embedded real-time systems is increasing rapidly: automotive electronics, office automation, telecommunication, etc., benefit from this progress with...
the consequence of more functionality and complexity at lower prices. Hardware/software
codesign has a key role in this development: a real-time system is no longer viewed as a
system consisting of hardware on the one side and software on the other side. Instead, the
system is described by its behavior, together with time constraints. We envision an
environment where the designer mainly concentrates on the specification—the hard-
ware/software trade-off is supported by a computer-aided, for the most part automated,
partitioning procedure and descriptions for both the hardware and software are generated.
Consequently, the design turnaround times can be minimized.

The architecture of a real-time system depends on parameters such as operating sys-
tems, scheduling algorithms, selection of a processor core, communication concepts, etc.
A starting point to approach hardware/software codesign is an important class of embed-
ded systems—single-chip microcontrollers. The configurable functionality offers a wide
field to test partitioning algorithms, optimization goals, etc. A loss in optimality is
compensated by much shorter design times, and, thus, the chance to optimize on higher
levels.

Recent trends support the investigations in hardware/software codesign: today’s micro-
controllers are often programmed in a high-level language, mostly C. That makes the
partitioning algorithms independent of the specific hardware, partitioning algorithms can
be formulated more abstractly, and, consequently, they are applicable to a larger area of
problems and architectures.

Today there are only a few approaches that try to automate the hardware/software
partitioning process, that is—to our minds—one of the key issues in hardware/software
codesign. The VULCAN system [1] and our COSYMA system (COSYthesis for eMbed-
ded Architectures) [2] do partitioning automatically under time constraints and optimization
goals. Other systems, such as [3], also approach this aim.

Both VULCAN and COSYMA have the same target architecture, small embedded
architectures, such as microcontrollers, consisting of a predefined processor kernel and
synthesized or user-defined coprocessors, memory, and peripheral units. Both use a
partitioning approach that iterates over synthesis and compilation. Iteration seems to be
necessary, because there are no approaches known that accurately estimate the results of
optimizing compilers and high-level synthesis tools with advanced techniques, such as
loop unfolding, tree height reduction, or percolation-based synthesis [4].

Whereas VULCAN is hardware-oriented, starting with an all-hardware solution (ex-
cept for data-dependent loops) and moves operations to software on a given processor core
until time constraints are violated, COSYMA is software-oriented starting with an all-
software solution on a given processor core and moves operations to peripheral hardware
until time constraints are no longer violated. An advantage of starting with an all-software
solution is that no restrictions on the system input description are necessary (dynamic data
structures, pointers, etc.). Still, it requires a somewhat more complex internal data struc-
ture and partitioning.1

Both approaches assume implicitly that synthesized hard-wired functions give faster
results than a software implementation.

There are other approaches viewing hardware/software codesign as a partitioning
problem: in [5], a clustering approach using closeness criteria (see [6]) controls the

---

1 This, of course, does not mean that all of these semantic constructs can be moved to hardware.
partitioning process where the user decides on the clustering. In [7] a speedup of a standard processor core (MC68010) is obtained by an instruction set metamorphosis. Code segments that are computationally intensive are moved to hardware, but this method is limited to the coarse granularity of the function level. The resulting high speedup values, however, seem unlikely for modern RISC processors.

Approaches working with manual partitioning are the CODES environment [8], which emphasizes the integrated design of hardware and software, specification, and cosimulation. A rapid prototyping approach for mechatronic system design is published in [9].

In the next section, a survey of the COSYMA system and its aims are given. Section 3 deals with estimation of the communication costs needed by the partitioning algorithm (Section 4). Then, Section 5 explains the way a coprocessor hardware is generated when the partitioning algorithm has proposed a codesign. In Section 6 our runtime analysis tool for hardware/software codesigns is described briefly. Finally, Section 7 shows some results obtained by an automatic partitioning process, and a conclusion is given in Section 8.

2 SOFTWARE-ORIENTED APPROACH TO HARDWARE/SOFTWARE CODESIGN

We will introduce our hardware/software codesign approach, which is based on the standard architecture consisting of a processor core, memory, and customized hardware.

The main intention is to implement as many system parts as possible in software running on the processor core. There are several reasons for this choice:

- High memory density
- Availability of optimally adapted compilers
- Careful verification and field test of standard hardware
- Declining synthesis efficiency for larger circuits
- High flexibility in case of modifications

External hardware is generated only when time constraints are violated, except for basic and inexpensive I/O functions and user-provided peripheral units, for example, selected from a library.

Whereas time constraints for interface control signals, such as Request and Acknowledge signals, concern a smaller part of the whole control task and, as such, leave little architectural choice, other timing constraints are more global, such as control process cycle times, dead times, data sampling rates, and process intercommunication. These more global constraints concern large code sections and allow many function implementation choices in hardware. If multiple tasks are executed concurrently, one might even decide to move part of another task to a hardware function to save processor time for the critical task.

The problem is to analyze the software and to extract an appropriate part of the software for implementation in hardware to meet timing constraints. An advantage of this approach is the flexibility in function selection. At this point, we would like to be able to use libraries, synthesis tools, and user experience. So, we would like to extract the following circuit types:
• **Primitive structures at the interfaces** (e.g., counters and timers). Currently, we assume synchronous circuit structures that can be implemented with a synthesis system. We also assume that interface functions asynchronous to the program flow are described as separate processes. Moreover, the user should be able to provide the hardware structure.

• **Coprocessors.** Coprocessors should be small, such that they can be synthesized by high-level synthesis.

• **Second core processor.** If a coprocessor is not appropriate because there is no distinct critical software function or it is too large, another (possibly different) standard core could be implemented.

In all three cases, the user must be able to support the extraction, that is, determine which parts should be implemented in hardware in case of, for example, predefined hardware components.

The automated extraction is a partitioning problem. Because analysis and extraction are done on the software functions, we call our approach *software-oriented* hardware/software codesign.

As mentioned in Section 1, C seems to emerge as an important language for programming-embedded control systems. So, we chose C as the input language to our COSYMA design system. Because we understand that software development, software efficiency, and verification are the dominant problems of control system design, we avoided to impose any limitations on the C language. By choosing a suitable internal data representation and translation, we were able to support the full C language, including dynamic data structures.

Because C has no notion of timing and no task concept, we added few language constructs for timing and task intercommunication. Timing can be defined as $t_{\text{min}}$, $t_{\text{max}}$, and $t_{\text{duration}}$ (see, e.g., [10]), where each timing constraint refers to two labels formulated in the C language. Again, these extensions are not a result of hardware/software codesign but a necessity for the description of control tasks. The syntax of these constructs is not significant and could be replaced by another real-time C dialect.

As required, the designer can provide hardware structures. The behavior must be described in a C function. This is well known from synthesis (e.g., [11]). The same is possible the other way round; that is, the designer may choose that a C-function must not be implemented in hardware to allow modifications until after hardware design. The resulting superset of C is called C*.

Figure 1 shows the COSYMA experimental system. A C* compiler translates the input description into an extended syntax graph, the ESG [12]. We decided not to use a data and control flow graph as internal representation, as usually employed in high-level synthesis systems, because it would be incomplete or inappropriate for representation of dynamic data structures, recurrence, and parallel processes and, thus, conflict with the software-oriented approach. The syntax graph is extended by the symbol table and by links showing the data dependencies to allow, for example, rapid estimation of communication costs. The ESG permits partitioning at the statement, basic block, and function levels. Additional details on the graph will be given later. There is a simulator for the ESG that allows simulation of a C* description (with parallel processes) and profiling. The profiling information is required for cost function computation and timing analysis.
The ESG parts to be implemented in software are translated back to C programs. To communicate with graph parts moved to hardware, communication macros are inserted. The protocol is generated from a template, whereas data to be exchanged are derived from a data flow analysis (next section).

Because of the similarity of C* and C, and the syntactical structure is maintained in the ESG, the original program structure can be rebuilt in this step. Thus, the system designer has direct control on the software structure, and a good programming style will be kept for compilation. As an additional advantage, the derived program is still understandable to the user, such that user interaction could be feasible. For compilation, we currently use the GNU C compiler, which can generate code for a variety of processor cores. As a beginning, we support a SPARC processor core [13].

Then, a runtime analysis checks if the timing constraints given in C* are met. We found that, to be accurate enough, complex processor cores with load–store architecture, pipelining, and concurrent units (or floating point coprocessors) require an analysis that models the data and control flow in the processor [14], which known approaches to runtime analysis (e.g., [15], [16]) cannot provide. In addition, hardware timing must be included. Experiments with a larger example, however, lead to RT-level simulation times of more than 0.5 hours on a SPARC 10/41 for each input pattern, even in the unpartitioned case. Iteration would be impractical under these circumstances. So, we developed a
hybrid approach combining RT-level simulation and formal analysis, giving results that, in our experiments with software timing analysis, deviate less than 1% from RT-level simulation with a few seconds of computation time per input pattern [14].

The ESG parts to be implemented in hardware, that is, coprocessor or peripheral hardware, are mapped to the hardware description language of the high-level synthesis system. The communication protocol is realized using templates with input and output statements. Currently, we use the high-level synthesis system OLYMPUS [11] from Stanford, which uses HardwareC as an input language. It provides an overloading mechanism, which allows us to use our own module library for HardwareC operators as well as user-defined hardware functions. Currently, our work is focused on partitioning of loop coprocessors, so we overload a library of high-speed arithmetic units. For SPARC coprocessors, we use only units with a word length of 32 bits. As output, OLYMPUS generates a gate-level description.

Because OLYMPUS synthesis and runtime analysis are computation-time intensive, the iterative partitioning uses a cost function based on estimated hardware cost and timing data. This is what we call the inner partitioning loop. In Figure 1, it is marked with a dotted surrounding box. The outer loop, including synthesis and runtime analysis, provides hardware cost and timing data. Memory costs are not yet included but they could be derived from the object code size and profiling data (dynamic memory allocation). The actual timing and costs are then used to adapt the estimated values of the partitioning cost function of the inner loop. Whereas the inner loop is fully implemented and operational, the outer loop (only this part) still requires some manual support. This, however, only means that the experiments take longer, are less optimal, and must be limited in size.

3 COMMUNICATION COST ESTIMATION

3.1 Internal Graph Representation Extended Syntax Graph

As mentioned above, our software-oriented approach makes it possible to handle dynamic data structures, pointers, recursion, and parallel processes. We are not able to implement all these elements of the C\textsuperscript{\textregistered} language in hardware, but the software of the generated system can contain all structures. A suitable data structure for the internal representation has to be able to describe the whole range of the C\textsuperscript{\textregistered} language.

In the area of high-level design, two types of representation are commonly used, the data flow graph (DFG) and the control and data flow graph. These graphs are not sufficient at completely describing dynamic data structures, because the data elements accompanying the pointer are often unknown at compilation time. In contrast, syntax graphs, for example, those used by MIMOLA, offer the possibility to describe all elements of the C\textsuperscript{\textregistered} language.

For the partitioning process, COSYMA requires information about the data dependencies. Using a regular syntax graph, this would be a time-consuming task. For that reason, we extended the usual syntax graph description by an implicit symbol table and local DFGs. We call this extension the ESG [12].

The ESG is a directed acyclic graph describing a sequence of declarations, definitions, and instructions. Special nodes, the so-called time nodes, describe user-defined time
constraints. Each identifier occurring in the graph is accompanied by an arc to its definition. Conversely, each definition is extended by pointers to all instances.

This implicit symbol table does not contain all information on data dependencies between operations that COSYMA requires to perform a local scheduling on the graph to estimate hardware execution times. Therefore, we overlaid the syntax graph by a second graph, consisting of cross-linked blocks, the so-called Basic Scheduling Blocks (BSBs). These BSBs contain local DFGs that are built using the operator nodes of the syntax graph. Since both graphs share the same operator nodes, a rapid transition between the two graphs is possible. Figure 2 illustrates a part of a syntax graph and the corresponding BSB. At the top of the BSB, a set \( i_{in} \) of nodes is figured attributed with a set of operands defined outside the BSB and used inside it. The nodes placed at the bottom of the BSB are attributed with the set of operands defined inside the BSBs, called \( i_{out} \). These two sets of nodes are used for estimating the costs for the communication between the hardware and software described later. Whereas the operand nodes in Figure 2b are different from the one in Figure 2a, the identity between the operator nodes is shown by identical node indices at the nodes.

### 3.2 Communication Cost Estimation Using the ESG

For the partitioning process, COSYMA has to estimate the communication costs between the processor core and a coprocessor. A coprocessor can execute several functions. Each function is built from a set of related BSBs chosen for hardware implementation. Executing one of the coprocessor functions, two types of variables have to be transferred between the processor and coprocessor:
1. Variables defined by the software and used by the hardware.
2. Variables defined by the hardware and used by the software.

We estimate the costs of communication by simply using the two sets of operand nodes of a BSB described above. Let $\mathcal{BB}$ be the set of extracted BSBs, which belong to one coprocessor function. The set of input variables $\text{in}_{\text{HW}}$ is calculated as follows:

$$\text{in}_{\text{HW}} = \bigcup_{b \in \mathcal{BB}} \left( \text{in}_a - \bigcap_{b \in \text{predecessors}(a)} \text{in}_b \cup \text{out}_k \right).$$

(1)

Comparing only the direct predecessors of a BSB, we do not take those variables into account that have been defined in a BSB that is also moved to hardware and is no direct predecessor of the viewed BSB. To avoid a time-consuming global data flow analysis we accept this, as we hope—negligible—error in our estimation. The calculation of $\text{out}_{\text{HW}}$ can be derived similarly. Currently, the cost function limits the partitioning to basic block granularity. Extensions to the statement and to the function level are planned.

The estimation of the communication cost takes place as part of the inner loop, which has to be executed very quickly. So, we relinquished an exact data flow analysis. To avoid redundant data exchange, we have to be more precise during hardware generation. Only the variables defined in software and used in hardware have to be transferred to the coprocessor. Conversely, only the variables defined in hardware and used by the software are transferred back to the processor. To establish the data dependencies we have to execute a global data flow analysis. Since the global data flow analysis is part of the outer loop and communication has a high impact on performance, as we see from the results, it is justified to spend time on it to avoid unnecessary communication.

The current target architecture is a single processor (SPARC) and a coprocessor. Communication between software and the coprocessor is done by memory coupling and a simple communicating sequential processes (CSP)-type protocol. That means both sides execute in mutual exclusion.

The result of the partitioning process described later is a graph with marked nodes. The nodes attributed with a mark are chosen for the extraction to hardware. Each continuous sequence of marked nodes is mapped to one coprocessor function. Currently, such an sequence area consists of one or several BSBs, but we are extending COSYMA to the extraction of single statements.

To distinguish the coprocessor functions, a BSB index is transferred from the software to the coprocessor, when it is started. The HardwareC process description is, therefore, structured as a "case" statement that is switched on the index. Depending on the index, the coprocessor reads the input variables from the common memory, commutes the chosen function, and writes the result back to the memory. Our communication mechanism does not allow the call of a software function from the hardware, because such a mutual context switch would be very time-consuming. That is why every function called by the hardware has to be extracted, too.

---

2 This is a very simple protocol, but we are not restricted to that.
4 HW/SW PARTITIONING

Hardware/software (HW/SW) codesign is viewed by us mainly as a partitioning problem. As described in [2], our partitioning goals are, in decreasing priority, meeting real-time constraints, minimizing hardware costs, and minimizing computation time for a design and, thus, the turnaround time.

Although the first goal has to be met under all circumstances (otherwise the system will not work correctly), the second and third objectives are real optimization goals, for example, a minimum has to be found. As shown later, one of the main problems in our partitioning problem will be the transformation of the real-time constraint into an optimization goal to get a uniform and simple cost function.

Hardware/software partitioning is a combinatorial optimization problem. Given an amount of hardware components [i.e., arithmetic logic unit (ALU), multiplier, etc.], our task is to find a minimum set of system functions (code segments) that leads to a sufficient speedup when implemented in hardware. The features of the simulated annealing optimization algorithm are as follows:

- **Flexibility in cost function:** The simulated annealing has been developed for general usage in combinatorial optimization problems. The cost function is application-specific.
- **The possibility of a quality/computation time trade-off:** By choosing parameters such as annealing speed, the user determines the quality and computation time of an optimization.
- **Well-investigated mathematical behavior**

To minimize the computation time of the annealing algorithm, three categories of improvements exist: parallel implementations, carefully controlled move generation strategies, and efficient annealing schedules. We implemented the algorithm described in [17], which belongs to the last group. It is based on a dynamic schedule; that is, control parameters are adapted automatically during the optimization process.

4.1 Implementation of Cost Function

A problem of our approach is that we start optimizing with a nonfeasible solution (an all-software solution) in contrast to other fields of application—for example, standard cell placement—where simulated annealing starts with a feasible solution. The reasons are hard limited time constraints (see above) in our problem. Our approach is to map the original problem consisting of a set of constraints (time constraints) and an optimization goal (hardware costs) to an optimization problem without constraints, but a highly weighted time cost. More precisely, the time cost is a function of the deviation from a given time constraint $T_C$ to the actual execution time $T_S$ of the whole real-time system. This optimization always starts with a feasible solution, but the time cost must be chosen such that simulated annealing is driven strongly into the direction of a feasible solution of the original problem.
Before going into more details of the components of our cost function, we will define the term **hardware extraction**: it is a specialized cost function that selects code sequences of the real-time system description that can be implemented efficiently\(^3\) in hardware. Examples are coprocessors, especially for loops; peripheral functions such as counters; and other microprocessors.

At the current time, we focus on extracting coprocessors for loops and time constraints for the whole system. Extraction of coprocessors is done by moving BSBs from software to hardware.

Our current cost function is presented in the following. First, we describe the cost estimation \(\Delta c\) for any single BSB that is transferred from software to hardware, and then we describe the cost function computation.

One goal is to minimize the amount of application-specific hardware. No explicit hardware costs appear in \(\Delta c\). Currently, the user defines the number of function units to be used, and we assume that these hardware components can be reused\(^4\) for all BSB sequences. Furthermore, the estimated communication overhead between the software and hardware components and vice versa should be taken into account. This leads to the estimated cost \(\Delta c\) of a single BSB, assuming that it is moved from software to hardware:

\[
\Delta c = a(T_c,T_S) (t_{\text{nff}} + t_{\text{com}} - t_{\text{HW}/SW} - t_{\text{SW}})i.
\]

where:

\(t_{\text{nff}}\) is the schedule time of a BSB. If the number \(n\) of hardware components (or sets of components, i.e., \(n = 1\), means there is only one ALU, one multiplier, etc.) is not limited, then \(t_{\text{n}}\) is equal to an ASAP schedule. If \(n\) is limited, then \(t_{\text{c}}\) is computed with a simple list schedule.

\(t_{\text{com}}\) is the sum of the data transfer times from software to hardware and vice versa. As already explained, it is taken into account that, in the case of extracting two or more sequential BSBs, the transfer time will be lower than the sum of the transfer times needed for each single BSB. Using the in and out sets of a BSB, communication cost calculation is reduced to simple set operations (see Section 5), thereby assuming a fixed computation time per transfer.

\(t_{\text{SW}}\) is the computation time needed for a software implementation of that BSB. Assuming that application-specific hardware is, in general, faster than the corresponding software implementation (our approach is based on this prerequisite), the term in brackets will be less than zero.\(^5\) We estimated this time by a simple table look-up assuming constant execution times for the assembler instructions not taking into consideration pipeline interlocks, etc. Because of great inaccuracy (in some cases up to 50%), we now use our precise hybrid timing analysis (HTA) tool (Section 6).

\(t_{\text{SW}/HW}\) is the overlap time between hardware and software execution. Because of our simple communication protocol, \(t_{\text{SW}/HW}\) amounts to zero.

---

\(^3\) Here: low area overhead at high speedup.

\(^4\) This can be changed when partitioning is improved.

\(^5\) In Section 7 we will discuss a counterexample.
expresses the number of times this BSB is executed during the whole runtime (obtained by profiling). \( \Delta c \) is multiplied with this factor, meaning that \( \Delta c \) is obtained each time the BSB is executed. As seen later, this results in extracting computationally intensive sequences with a higher probability than BSBs executed a single time.

\[
e_{c}(T_{c}, T_{s}) = \text{sign}(T_{c} - T_{s})e^{cT_{c}-T_{s}}T,
\]

with \( T_{c} \) the given time constraint, \( T_{s} \) the resulting time needed by the hardware/software system between the time labels of \( T_{c} \), and \( T \) a constant factor.

This corresponds to an exponential weighting of runtimes above the given constraints. Below the constraints, the sign is changed to avoid that basic blocks are moved unnecessarily to hardware, increasing the synthesis task and hardware (controller) cost.

With the increments \( \Delta c \) and \( \Delta t \) [where \( \Delta t \) denotes the term in brackets in Equation (2), the cost function \( C \) and the time \( T_{\text{system}} \) are updated as follows, assuming \( C_{-1} \) refers to the last annealing move]:

\[
C = C_{-1} \pm \Delta c
\]

\[
T_{s} = T_{s-1} \mp \Delta t,
\]

where the upper operators refer to an \( SW \rightarrow HW \) move and the lower operators to an \( HW \rightarrow SW \) move.

In Section 7, we will see that the partitioning algorithm is suited to extract computationally time intensive BSBs, fulfilling the real-time constraint \( T_{c} \) when possible while running in an acceptable time. The results required up to 100,000 moves controlled by the annealing schedule and none took more than 30 s of optimization time (user time) on a SPARC1 + workstation.

4.2 Behavior of Simulated Annealing

Figures 3 and 4 demonstrate the functionality of our cost function. The horizontal axis in each figure shows the number of BSBs. The vertical axes in the upper portions of Figures 3 and 4 show how many times a specific BSB has been moved to hardware (for 10 partitioning runs), whereas the vertical axes in the lower portions of Figures 3 and 4 show how many times a BSB was iterated. For every benchmark, two graphics are shown:

- First 10 partitioning runs were executed, each starting with a different random seed. In these graphics, there is shown how often each software component was transferred to hardware. The interesting aspect is the repeatability and selectivity of our cost function.
- The picture below shows how many times a BSB has been executed during a simulation, demonstrating the selective extraction of computation-intensive code segments.
As seen in the upper portions of Figures 3 and 4, most extracted code segments directly correspond to our computation-intensive code segments. However, there are also cases in which code segments have never been transferred to hardware, reflecting high communication costs; for example, the single BSB might be executed in the form of a hardware realization in a shorter time than in software realization. However, the communication has been so large that a hardware solution would have been too time-intensive. So, the time penalty is multiplied by the number of iterations, making a move to hardware very unlikely. That is exactly what the cost function is suited for. In addition, the repeatability is very high: in the upper portion of Figure 3, all BSBs are transferred to hardware in all optimization runs.

5 TRANSLATION TO HARDWARE DESCRIPTION

Based on the BSBs that have been marked by the partitioning process, a translation to an equivalent hardware description is done. General problems concerning a translation from

---

6 At the current time we use HardwareC and the OLYMPUS system for synthesis, but we are not restricted to that; we are working on the implementation of our own synthesis system, BSS.
a software part to a hardware description are due to different features of software languages and hardware description languages, especially concerning the data structures; and different semantics of hardware and software languages.

We will concentrate on the first aspect because of its importance in our software-oriented approach in hardware/software codesign. The differences listed below are not only fixed to the current HDL we are using (HardwareC), but they might also cause problems when using another HDL. The main differences are as follows:

- **Data types.** HDLs for synthesis are limited mostly to simple data types such as single bits or integer variables. Our input language is a superset of the C language. HardwareC has no floating point types. Consequently, operations on floating point values have to be mapped COSYMA-defined or user-defined hardware units and the floating point values have to be expressed in an integer format. Similar problems occur for pointers, structs, and unions.
- **Subroutines.** Whole subroutines can be transferred from software to hardware with some modification: for those subroutines that return a value, variables have to be inserted explicitly. A global data flow analysis is required to identify side effects of a subroutine, such as access to global variables, to identify the required communication.
Another meaningful limitation in subroutine translation concerns the granularity: if a whole subroutine except, for example, one statement would be moved to hardware, a software call from the hardware side would be initiated. This would raise the communication overhead. Therefore, those subroutine calls are totally implemented in hardware.

Other high-level synthesis systems are not able to handle procedure calls. That means a flattening has to be done. Or the whole function can be scheduled and implemented as a separate hardware unit.

This should give only a short impression of the long list of differences between software and hardware description languages that complicate and, in some cases, prohibit a translation to hardware, but we will not focus on this problem.

Currently, granularity in hardware extraction is based on the BSBs: At least one BSB—normally a set of sequential BSBs, called segments—is moved to hardware. The syntactical correctness by moving a segment from software to hardware is guaranteed by the definition of a BSB (see Section 3.1). Semantical correctness is arranged by a local\footnote{For example, only concerning the code segment.} data flow analysis. Let use[S] be the set of variables used within segment $S$ and def[S] be the set of variables to which a new value has been assigned in segment $S$. Then

\[ \text{def. bu}[S] = \text{def}[S] - (\text{def}[S] \cap \text{use}[S]) \]

(6)

denotes the set of variables to which a value has been assigned before its usage in segment $S$. This is identical to the set of variables that has to be sent from hardware to software, assuming that segment $S$ has been implemented in hardware. The set use[S] is an upper bound for the number of registers that have to be allocated for the hardware implementation of segment $S$, which is determined according to Section 3.2.

The next step is the instantiation of an appropriate hardware template. Figure 5 shows a state transition diagram of such a template: in the state wait, the coprocessor is waiting until it receives the signal from the software side to take control. Then the state select selects one of the states $S1$ to $SN$, where the parameters $1$ to $N$ have also been transferred from the software side. Each of $S1$ to $SN$ contains one of the segments discussed earlier. That means in most cases more than one state is hidden in each of $S1$ to $SN$. In the switchback state, control is given back to software, and the coprocessor is waiting again for execution of a new segment.

The last step of instantiation consists of the communication protocol implementation (see in Section 3). Data objects have to be transferred from the software side to the main memory to make sure they are not stored in registers. The coprocessor hardware loads them into internal registers and writes them back to the main memory after execution where the results can be accessed by the software.

An extraction of a generated example for both—hardware and software parts—is given in Figures 6 and 7. Whereas Figure 6 represents the software solution of a small loop, Figure 7 shows the same software part after the partitioning process, and in Figure 8 the corresponding hardware template including an adequate hardware description of that loop is listed.
Parts I and III in Figures 7 and 8 represent the realization of the communication protocol, for example, data transfers from and to main memory. Part II in Figure 7 is the coprocessor call (assembly language function) that triggers the coprocessor to execute the corresponding hardware routine. In Figure 8, part IV is the realization of the switch state from where a call to the execution part (part III in Figure 8) is done. This communication template is inserted on the C-source level.

```c
for (j=0; j<kernel_cols; j++) {
    sumval += IN[(row+i)*256+(column+j)] * FIL[(i)*3+j]);
}
```

Figure 6. Software segment before partitioning.
Figure 7. Software segment after partitioning.

6 SIMULATION

After the partitioning process, we have to check whether the time constraints are met.

The computation time estimation presented in Section 4 is based on the ESG, which represents the C8 input program. That is the reason why compiler optimization effects and processor architecture are not taken into consideration. Modern RISC architectures with pipeline structures and—at least partially—data-dependent instruction execution times require an exact simulation to evaluate the execution time. At the current time, such an exact analysis is a time-consuming task. Formal approaches [18] can be used to cut the software analysis time; however, there are some sources of imprecision that are not considered in these approaches. Instruction execution in RISC processors is a complex interplay of several functional units. In particular, pipelining leads to a program-dependent instruction execution times through pipeline interlocks. Therefore, RT-level simulation seems to be the only way; such is used in [19].

The SPARC architecture is still rather simple compared with newer superscalar architectures with score-missing, such as the new Motorola 88100 [20]. There is little hope that there will be a precise assembly or programming-level timing estimation suitable to fine-grain partitioning for pipelined and superscalar RISC processors.

The problem of simulation times compared with formal analysis is repeated execution of the same statements in loops and function calls on the one hand and in repeated runs of the simulation model with different input patterns on the other. In the context of hardware/software partitioning, it is always difficult to define one single, worst-case input pattern, because when a time-critical part is moved to hardware, the worst-case input pattern might have changed.

To overcome repeated execution, we developed an approach, where each part of a
static x[4]=0;
static y=0;
/* waiting for start signal */
/* this has to be 1 cycle */
do <
y=read(startbit);
write ready=1;
x=read_from_sw( ACTIVATE_ADR, adress, data, dn, rd, we ) ;
>
while(1) ;
/* the sparc writes to a
certain adress, which activates the hardware */
/* switch to the specific routine */
<
write ready=0;
switch( x ){
  case 1:
    hw_proc_1(adress, data, dn, rd, we);
    break;
  default:
    hw_proc_1(adress, data, dn, rd, we);
    break;
  }
  I
twrite ready=1;
  write ready=0;
  I
/* *******START of #1*********/
procodure hw_proc_1(adress, data, dn, rd, we)
inout port data[32];
out port adress[32];
out port rd[1];
out port we[1];
out port dn[1];
  boolean FIL_1[32] ;
  boolean IN_1[32];
  boolean column_1[32];
  boolean i_1[32];
  boolean row_1[32];
  boolean summval_1[32];
  boolean temp_var_1_1[32];
  boolean temp_var_2_1[32];
/* --START: communication calls ------- */
{ FIL_1 = read_from_sw( 2048, adress, data, dn, rd, we ) ; } 
{ IN_1 = read_from_sw( 2052, adress, data, dn, rd, we ) ; } 
{ column_1 = read_from_sw( 2056, adress, data, dn, rd, we ) ; } 
{ i_1 = read_from_sw( 2060, adress, data, dn, rd, we ) ; } 
{ row_1 = read_from_sw( 2068, adress, data, dn, rd, we ) ; } 
{ summval_1 = read_from_sw( 2072, adress, data, dn, rd, we ) ; } 
/* --END: communication calls -------- */

/* *******START of #1*********/
for( i = 0 to kernel_cols 1 ) do 
{ temp_var_1_1 = ARRAY_READXIN_1+(d*(row_1+i*1+256+column_1+i*1));
  temp_var_2_1 = ARRAY_READX( FIL_1+i*1+(0*1+1*1));
  summval_1 = summval_1+temp_var_1_1*temp_var_2_1;
} 
/* -- hw to sw communication ---- */
{ write_to_sw( summval_1, 2052, adress, data, dn, rd, we ) ; }
/* *******END of #1***********/

Figure 8. Hardware template with extracted segment.
Table 1. Partitioned Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Used Clock Cycles</th>
<th>( t_c ) [%]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>diesel</td>
<td>22 403</td>
<td>16 394</td>
<td>9.9</td>
</tr>
<tr>
<td>smooth</td>
<td>1 781 712</td>
<td>1 393 525</td>
<td>49.6</td>
</tr>
<tr>
<td>third</td>
<td>1377</td>
<td>1514</td>
<td>13.8</td>
</tr>
</tbody>
</table>

The program is simulated once in the beginning and formal analysis is used to derive the total execution time when partitioning occurs. Using a set of user-given input patterns, we establish profiling information for each loop and function. Under consideration of pipeline overlaps, we compute the program execution time with the aim of this profiling information [14].

Table 1 shows the simulation times of four benchmarks. First is an HDTV-Chromakey-Algorithm [21], which consists of 1400 lines of C code. The second algorithm, a digital controller of a turbocharged diesel engine, contains a loop executed 50 times. The third benchmark is an automatically configured fuzzy controller. The last one, a quicksort algorithm, is a simple example of interdependent execution of nested loops.

For evaluation of the hardware execution time, a combination of profiling data and the scheduling information, derived from the high-level synthesis, are required. The algorithm and details of implementation are described in [14].

Currently, we can provide only data on the unpartitioned case, but the extension on the partitioned system is under implementation.

7 RESULTS

For the following results, it is assumed that hardware and software parts are executed using the same instruction cycle time. We are not limited to this assumption, but it simplifies the comparison of the execution times (before and after partitioning). The cycle times were calculated by the HTA tool in connection with the high-level synthesis system OLYMPUS: first, the all-software implementations of the benchmarks were derived from RT-level simulation. After the partitioning process, hardware descriptions with the same structure as shown in Figure 8 were generated, and then the hardware execution times were simulated with MERCURY, an OLYMPUS tool, and written back to the ESG. So our HTA tool could read this information and analyze the common hardware/software execution time (column HW/SW in Table 1).

To keep synthesizing and (hardware-) simulation times low, we restricted ourselves to design points aiming at a speedup of ≈2.

First, we took the benchmark named diesel (see Table 2), a real-time algorithm for the digital control of a turbocharged diesel engine. Our timing analysis tool showed a computation time of 22,403 cycles when running on a SPARC1 + processor. An automatically generated hardware/software codesign could reach a speedup of 1.4 (16,394 cycles). As
Table 2. Execution Times of the Simulation for a Single-Process Execution on a SPARC

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Simulation</th>
<th>With Profiling Information</th>
<th>Worst Case</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>1087.1</td>
<td>1.7</td>
<td>1.6</td>
<td>with optimization</td>
</tr>
<tr>
<td>Diesel</td>
<td>2688.4</td>
<td>2.0</td>
<td>2.0</td>
<td>without optimization</td>
</tr>
<tr>
<td>Fuzzy</td>
<td>1.4</td>
<td>0.1</td>
<td>0.1</td>
<td>with optimization</td>
</tr>
<tr>
<td>Quicksort</td>
<td>1.7</td>
<td>0.1</td>
<td>0.1</td>
<td>without optimization</td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>0.9</td>
<td>1.0</td>
<td>with optimization</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>0.9</td>
<td>0.9</td>
<td>without optimization</td>
</tr>
<tr>
<td></td>
<td>0.4</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>with optimization</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>without optimization</td>
</tr>
</tbody>
</table>

Table 2 shows, only one application-specific set of hardware components (one 32 ALU that consists of $74 \times 181$ ALUs and a multiplier requiring two instruction cycles for a 32-bit multiplication) was used in high-level synthesis, where no chaining or pipelining is used. Another benchmark executing a filter algorithm on a digital image gave almost the same result.

An interesting example to show that automated hardware/software codesign is not a trivial task is demonstrated using the third benchmark. At the same conditions where diesel and smooth resulted in a real speedup, we reached a “speedup” of 0.9. Here the optimizing potential of the GNU C-compiler was not taken into consideration by the partitioning tool. Interpretation of partitioning effects in the COSYMA system is done in more detail in [22].

8 CONCLUSION

We presented the current state of our COSYMA experimental system for hardware/software codesign. As shown, the simulated annealing heuristic seems to be a feasible approach to automate the partitioning process. We explained how a convergence under time constraints and optimization goals can be reached and—nevertheless—the cost function can be kept simple, consuming only a small amount of computation time.

The results presented are not yet optimal concerning the speedup but have been gained fully automatically. One of the reasons is that the outer loop still must be closed. Our future work will concentrate on improving estimation during optimization, choosing a better (i.e., faster) communication protocol, and implementing other high-level synthesis systems.

The supporting tools for hardware/software codesign are crucial for a practical system. We presented in short terms a hybrid timing analysis tool for calculating software execution times with a high precision at a fraction of the computation time that current tools need.
REFERENCES


[22] J. Henkel, T. Benner, and R. Ernst, "Hardware generation and partitioning effects in the
Jörg Henkel is pursuing a Ph.D. in Electrical Engineering from the Technical University of Braunschweig, Germany. For about 2½ years he has been involved in the development of the experimental system COSYMA for hardware/software co-design. His interests include hardware/software co-design, high-level synthesis and, computer architecture. He holds a diploma in Electrical Engineering from the Technical University of Braunschweig.

Thomas Benner is a research assistant at the Technical University of Braunschweig, Germany. His interests include hardware/software co-design, computer architecture, and digital circuit design. He holds a diploma of Computer Science from the Technical University of Braunschweig and is pursuing a Ph.D. in Electrical Engineering. He is a member of the German GI (Society of Computer Science).

Rolf Ernst is a Professor of Electrical Engineering at the Technical University of Braunschweig, Germany, where his interests are VLSI CAD and digital circuit design. Previously, he was a member of the technical staff in the CAD & Test Laboratory at AT&T Bell Laboratories and research assistant at the University of Erlangen in Germany. He holds a diploma in Computer Science and a Ph.D. in Electrical Engineering from the University of Erlangen. He is a member of the IEEE, the IEEE Computer Society, and the German GI.

Wei Ye is a Ph.D. candidate in Computer Science from the Technical University of Braunschweig, Germany. His interests include timing analysis of real-time systems, hardware/software co-design, and high-level synthesis. He holds a diploma in Computer Science from the Huazhong University of Science and Technology, Wuhan, China.
Nikola Serafimov received the M.Sc. degree in Electrical Engineering from the Technical University–Sofia, Bulgaria in 1988. From 1988 to 1992, he was with the Faculty for Telecommunications and Technologies at the same university as an assistant professor. Currently he is with the Institut für Datenverarbeitungsanlagen at the Technical University of Braunschweig, Germany and is working towards a Ph.D. dissertation. His interests include digital signal processing, high-level synthesis, design of communicating systems, hardware/software codesign.

Gernot Glawe is pursuing a diploma in Electrical Engineering from the Technical University of Braunschweig, Germany. His interests include hardware/software codesign and high-level synthesis.