Hardware-Software Cosynthesis for Microcontrollers

ROLF ERNST
JÖRG HENKEL
THOMAS BENNER
Technical University
of Braunschweig

The authors present a software-oriented approach to hardware-software partitioning, which avoids restrictions on the software semantics, and an iterative partitioning process based on "hardware extraction" controlled by a cost function. This process is used in Cosyma, an experimental cosynthesis system for embedded controllers. As an example, the authors demonstrate the extraction of coprocessors for loops. They present results for several benchmark designs.

SMALL EMBEDDED-CONTROL systems consisting of a few integrated circuits are a growing field with a large share of the semiconductor market. Applications include office automation, telecommunications, consumer products, and industrial and automotive control. Embedded control requires reactive systems—that is, systems that react in real time to external asynchronous events, rather than process an input and produce an output after some time, as in classical data processing.

An embedded-control system's architecture is a combination of programmable microprocessor cores with memory and hardwired or field-programmable peripheral devices. Hardware and software together form the control system.

Applications of small embedded-control systems are increasingly complex; examples include 3D signal processing, computer vision, and fuzzy logic. Consequently, the architectures have become more complex, catching up with workstation technology using 32-bit RISC (reduced instruction-set computer) processors.

At the same time, manufacturers have a strong incentive to speed up system design to meet tight time-to-market requirements. Hardware design often must start when the specification is still subject to change. All this makes designing the system increasingly difficult. As a result, small embedded-controller design is changing. At the high end particularly higher level languages (often C) are gradually replacing assembly coding.

For our research work, we selected integrated embedded systems, or microcontrollers, because of their manageable size and economic importance. To minimize customized hardware in microcontrollers, hardware designers are currently developing libraries of standardized peripheral components—for example, in the European OMI (Open Microcontroller Initiative) project. Although this approach allows fast design turnaround and quick modifications, it severely limits design space. At the controller interface, the library approach might be acceptably efficient because most interface functions are relatively simple (counters, timers, serial-parallel conversion), or they are standardized (CAN-Bus, ISDN, Ethernet) or analog (A/D conversion). Much more difficult, however, is

0740-7475/93/1200-0064$03.00 © 1993 IEEE

deciding which processor core(s) to use, whether to use one or more—possibly different—cores, and how to distribute the work load. Application-specific coprocessors could be very cost effective if they were targeted to those small parts of the software where most of the computation time is spent.

All these decisions require intricate knowledge of the system, which a hardware designer usually does not have, and they must be reevaluated in case of modifications. The library approach covers none of this. So, in general, the designer will stay on the safe side and overdesign processor performance, even in cost-sensitive volume markets.

A microcontroller overdesign can have a high impact on chip area. For example, the difference between a 32-bit RISC and a 16-bit processor may be several hundred thousand transistors, including additional memory for increased instruction and program size. So, for embedded systems, hardware-software codesign potentially has a much higher impact than, for example, logic synthesis.

In the following discussion, we use the term *hardware-software cosynthesis* for codesign systems aiming at automated cost optimization under constraints, mainly timing constraints.

A software-oriented cosynthesis approach

Our hardware-software cosynthesis approach is based on the standard microcontroller architecture, consisting of a processor core, memory, and customized hardware. The processor core is a standard microprocessor, and the customized hardware is synthesized (or user defined).

We implement as many operations as possible in software running on the processor core. The reasons for this choice include the high memory density of standard microprocessors, the availability of optimally adapted compilers, and the careful verification and field testing of standard cores. Moreover, it makes

For embedded systems, hardware-software codesign potentially has a much higher impact than, for example, logic synthesis.

software debugging simpler and overcomes problems of hardware synthesis efficiency for larger functions. Last but not least, it gives us much flexibility in case of modifications.

We generate external hardware only when timing constraints are violated. Exceptions are basic and inexpensive I/O functions—for example, the standard processor interface (address bus, data bus, and control signals), serial and parallel I/O, and user-provided peripheral functions such as an optimized field bus interface selected from a library.

Timing constraints for interface control signals, such as Request and Acknowledge signals, span a small part of the whole control task and thus leave little architectural choice. But other timing constraints are more global, such as control process cycle times, dead times, data-sampling rates, and interprocess communication. These more global constraints concern extended code sections and give much choice as to which part of a function to implement in hardware. If multiple tasks execute concurrently, one might even decide to move part of a noncritical task to a hardware function to save processor time for a critical task.

The problem is to analyze the software and to select an appropriate part of the software for implementation in hardware, to meet timing constraints.

At present, we can handle only coprocessors and user-defined interface modules. Eventually, we would like to use the following circuit types:

- *Primitive structures at the interfaces*: counters, timers, and so on. The user should be able to provide more complex peripheral structures such as bus interfaces.
- *Coprocessors*. Coprocessors should be small so that they can be implemented by high-level synthesis.
- *Second core processor*. If a coprocessor is not appropriate because there is no distinct critical software function or because it is too large, another (possibly different) standard core could be implemented.

In all three cases, the user must be able to define hardware modules. Hardware function selection and circuit type definition constitute a partitioning problem. Because analysis and partitioning occur in the software functions, we call our approach software-oriented hardware-software partitioning.

Related work

Srivastava and Brodersen present a CAD framework for rapid prototyping.[1] The target architecture consists of dedicated and programmable hardware modules, and the system software is generated to run on it. As in the Codes environment,[2] Srivastava and Brodersen emphasize integrated design of hardware and software, specification, and cosimulation. Windirsch et al. describe rapid prototyping in mechatronic system design.[3]

A second group of researchers views hardware-software codesign mainly as a partitioning problem. Barros and Rosenstiel[4] present a clustering approach using closeness criteria (see, for example, Lagnese and Thomas[5]) to control the partitioning process. The designer decides on the clustering. This indirect approach covers part of the design space. Athanas and Silverman use an "instruction set meta-

Finer-grain partitioning, using coprocessors and second cores, becomes more and more important as processor performance rises and system software increases.

morphosis" to speed up a standard processor core (MC68010).[6] Computation-intensive code segments are moved to hardware, but the method is limited to the coarse granularity of the function level. The resulting high speedup values, however, seem unlikely for modern RISC processors. All these approaches work with manual partitioning.

Only one system—besides Cosyma, the system we describe in this article—performs an automatic partitioning process.[7,8] (Woo, Wolf, and Dunlop's work heads in the same direction.[9]) This cosynthesis system, Vulcan, uses a system architecture similar to the one in our approach. The input language is HardwareC, a subset of C defined for hardware description, with integer as the only data type. The cosynthesis starts with a configuration in which all functions, except program constructs with unbounded delay, are implemented as hardware modules. The remaining functions are implemented as software on a standard core processor. Then the design system tries to gradually move hardware functions to software, checking timing constraints and synchronism as it does so. The high-level synthesis system Olympus generates the hardware. This is a hardware-oriented partitioning approach. In contrast to our software-oriented approach, only constructs that initially can be implemented in customized hardware can move to software. That means the input system description has some limitations in dynamic data structures and system complexity.

Microcontroller system modeling

Because C seems to be a preferred language for embedded-control programming, we pragmatically defined C^x, a superset of the ANSI C standard, as the input language. We avoided restrictions when we enhanced the C language, because software development, software efficiency, and verification are becoming dominant problems of embedded-system design, accounting for most of the development costs. Obviously, some C constructs cannot be mapped to hardwired logic, such as dynamic data structures, but in a software-oriented approach, this only means that these constructs must be excluded from implementation in hardware. Our main extensions of C are

- timing: minimum and maximum delays and duration between C labels of a task[10]
- task concept
- task intercommunication

Our approach to user interaction in the selection of hardware and the inclusion of user-defined hardware functions, such as library functions, is similar to the solution in the Olympus synthesis system.[11] The designer must describe the behavior to be implemented in hardware as a C function, which is then moved to hardware and implemented by synthesis or by the user-defined hardware. The same is possible the other way around; that is, the designer can define a C function that must not be implemented in hardware, to allow modifications even after hardware development.

Hardware-software partitioning problem

Partitioning must identify if and where system constraints, in our case timing constraints, are violated. Partitioning can occur at different levels of granularity: task, function, basic block, or even single statement. Partitioning on even lower levels, such as the assembly language level, is not useful because the assembly code is already based on processor details. In the following discussion we use the term *coarse-grain partitioning* for task-level and function-level partitioning and *fine-grain partitioning* for basic-block-level and statement-level partitioning. By these terms, Srivastava and Brodersen,[1] Buchenrieder and Veith,[2] and Athanas and Silverman[6] use coarse-grain manual partitioning approaches. Barros and Rosenstiel[4] use fine-grain manual partitioning, and Gupta and De Micheli[7] use fine-grain automatic partitioning.

Partitioning at the task or subtask levels is typical for manual design. Sometimes, partitioning at this level is almost obvious. An example is a signal-processing task consisting of high-speed filtering of input data exceeding a processor's performance, followed by a more complex algorithm with lower performance requirements.

Finer-grain partitioning, using coprocessors and second cores, becomes more and more important as processor performance rises and system software increases. At finer granularity, however, partitioning is less obvious and more difficult because its side effects have a high impact. The most important side effects are

- *Communication time overhead*: Additional I/O operations of the processor core require additional computation time. Load/store architectures typical of RISC controllers even require an extra instruction for each read and write operation. In an extreme case, the

overall timing might be worse than before partitioning.

- *Communication area overhead:* Besides obvious wiring overhead, communication can require buffers or memories. Buffer or memory size estimation is not always a simple problem.
- *Interlocks:* If variables are allocated to an external hardware register, they might not (yet) be available by the time the processor software can process them. This leads to waiting time in the software.
- *Compiler effects:* When a program is fragmented by the extraction of statements or basic blocks, the efficiency of compiler optimization will change. Also, pipeline efficiency and concurrent unit utilization (superscalar architectures) will be different. These effects are hard to predict.

In addition to the large design space including processor selection, peripheral component and coprocessor definition, and synthesis, which we pointed out earlier, these side effects make it even harder for a system designer to partition at levels of finer granularity. There are a few exceptions such as floating-point or graphics coprocessors.

Nevertheless, fine-grain partitioning offers a high potential for system optimization, as we will show in our examples. The exploitation of fine-grain partitioning is one opportunity provided by hardware-software cosynthesis. Therefore, our approach concentrates on fine-grain partitioning (currently on the basic-block-level only), but we can also use it for coarse-grain partitioning.

The Cosyma system

As a platform for our research, we developed the cosynthesis system Cosyma (cosynthesis for embedded architectures). Figure 1 gives an overview. The system description in C^x is translated into an internal graph representation

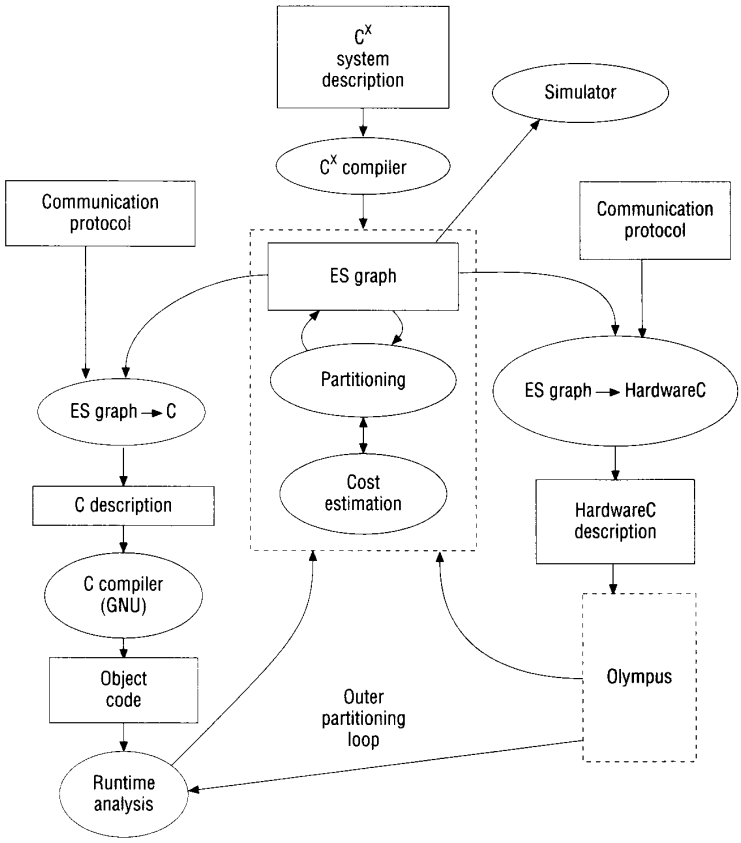


Figure 1. *The Cosyma system.*

suitable for partitioning. The following are the requirements of this internal representation:

- It should completely represent all input constructs including dynamic data structures, recurrence, parallel processes, and timing.
- The user should have strong influence on the syntactic structure of the software (to maintain good programming style).
- The representation should support partitioning and generation of a hardware description for parts moved to hardware.
- Estimation techniques such as a simple runtime estimation by local scheduling on the graph should be possible.

A control and dataflow graph, typically used in high-level synthesis, does not meet the first and second requirements but is appropriate for the last two. Therefore, we defined an extended syntax graph, or ES graph, which is a syntax graph extended by a symbol table and local data and control dependencies.[12] The ES graph is a directed acyclic graph describing a sequence of declarations, definitions, and statements. Each identifier occurring in the graph is accompanied by a pointer to its definition. Conversely, pointers to all instances extend each definition, building an implic-

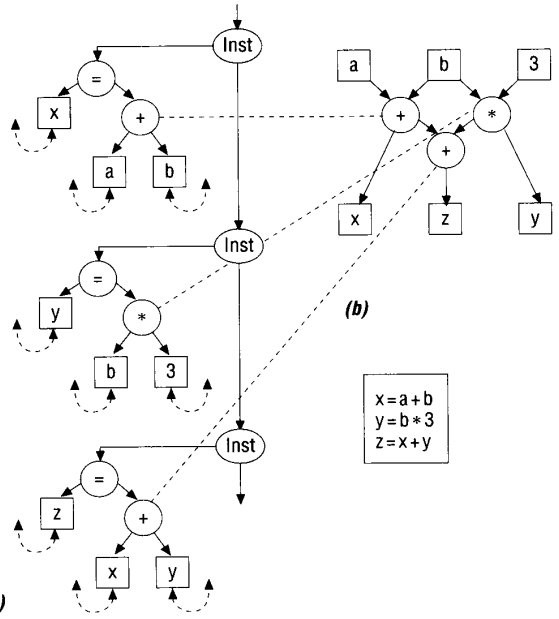


Figure 2. *The extended-syntax graph (ES graph): part of the syntax graph* ***(a)*** *and corresponding BSB* ***(b)****.*

it symbol table.

The syntax graph itself allows the description of the whole input language but does not contain any information about the data dependencies occurring in the graph. Therefore, we overlaid the syntax graph with a second graph consisting of cross-linked blocks, called basic scheduling blocks (BSBs). Both graphs share the same operator nodes, thereby enabling a fast transition from the syntax graph to the dataflow graph and vice versa.

Figure 2 illustrates the relationships between the two representations. The dashed lines represent the physical identity of the operator nodes in both graphs. The ES graph is implemented as a C++ class and appears as an overlay of a syntax graph by a control and dataflow graph. Thus, for a given node, the system can easily and quickly switch from one view to the other. A simulator for the ES graph supports verification of the C^x description (including parallel processes) and profiling. Profiling is required for partitioning.

We execute hardware-software partitioning on the ES graph by marking nodes to be moved to hardware. A translator (ES graph → C) generates C functions for the software, reconstructing the structure of the original C^x description preserved in the ES graph. Then, the hardware-software communication protocol is inserted. The communication protocol is generated from a template, and the data to be communicated are determined from a dataflow analysis of the ES graph.

A standard (GNU) C compiler generates the object code, which can then be simulated with an RT- (register-transfer) level simulator.[13] Currently, Cosyma supports only a Sparc processor core. We chose Sparc because it is becoming one of the preferred 32-bit RISC architectures in microcontrollers. We execute a runtime analysis on the object code to check for violations of the timing constraints in the C^x description. This runtime analysis can be an RT-level simulation, but we have also developed a hybrid timing verification approach that is much faster and provides almost the same precision.[13]

For hardware generation, Cosyma currently uses the Olympus high-level synthesis system.[11] Olympus accepts HardwareC as an input language. An important feature in this context is that Olympus allows us to link user-defined hardware modules to the synthesis process, either as HardwareC functions or by overloading the regular HardwareC operators. We overloaded our own library of 32-bit multipliers and ALUs.

Several independent but sequential subgraphs of the ES graph (statements, basic blocks, or functions) can be mapped to a single coprocessor. Therefore, if the coprocessor is activated, a subgraph index must be communicated from the processor to the coprocessor, indicating which subgraph to execute. The HardwareC descriptions for the individual subgraphs are encapsulated in a switch statement controlled by the subgraph index, BSB-Id (see Figure 3).

So far, we have not determined the hardware-software partitioning approach. Cosyma is not restricted to a particular approach; rather it was intended as an experimental platform for hardware-software partitioning approaches. However, there are some reasons to focus on iterative partitioning:

- The results of optimizing compilation and processor pipeline utilization are hard to predict. Thus, software timing estimation is very difficult, particularly if the partitioning approach must be usable for different processor cores and compilers.
- Estimating high-level synthesis results is even more difficult. The effects and applicability of high-level transformations, such as tree height reduction or percolation-based synthesis,[14] and the efficiency of scheduling and allocation are extremely hard to predict.
- Communication time overhead can

be high compared to circuit partitioning and can require up to hundreds of clock cycles for a single coprocessor run. This overhead depends on communication mechanisms, memory organization, variable allocation, and so on. So the overall costs of hardware-software partitioning can be highly nonmonotonic.

We concluded that an iterative partitioning approach would be best suited for cosynthesis with cost optimization. In our case, iteration includes hardware synthesis, compilation, and timing analysis of the resulting hardware-software system with RT-level timing precision. The iteration loop is shown as the outer partitioning loop in Figure 1.

For the partitioning process, we concentrate on stochastic algorithms. Stochastic algorithms let us use arbitrary cost functions and iteration steps to make a trade-off between computation time and result quality. Thus, they are well suited to partitioning experiments, even if the relation of computation time to quality might not be optimal. At present, we use simulated annealing.

Simulated annealing with each move evaluated throughout the design loop, however, would be impractical, considering the computation time for synthesis, compilation, and runtime analysis. Therefore, we introduced a dual-loop approach. We execute simulated annealing on an inner loop, based on a cost function with estimated results. This cost function is adapted to the actual results in the outer loop.

The cost function plays an important role in our partitioning approach. We use it not only to estimate costs but also to control the partitioning process.

Usually, simulated annealing starts with a feasible solution and accepts only moves that lead to another feasible solution. In our software-oriented approach, however, we must start with a nonfeasible solution—that is, a solution that does not meet the time constraints. Instead of trying to enforce a feasible solution as a starting point (as in the hardware-oriented approach), we solve the optimization problem with a high cost penalty for runtimes exceeding the time constraints and a steep decrease of costs for improved timing. This prevents annealing before a feasible solution is reached or further partitioning is not possible.

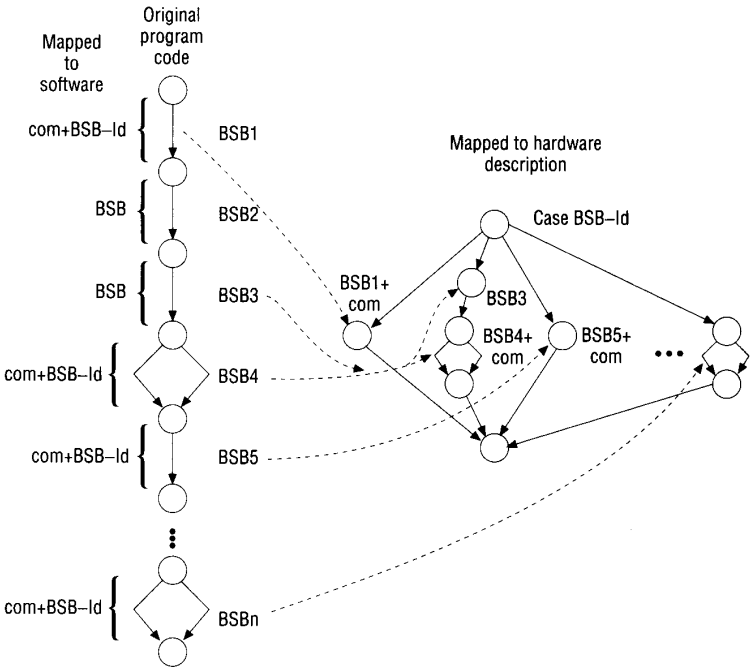


Figure 3. *Extraction of BSBs.*

Currently, all the tools we have presented are operational. Thus, we can achieve one fully automatic partitioning result. The automatic cost function adaptation, however, is not implemented yet. So if a first design does not meet constraints, we must start a new run through the Cosyma system. The user selects new parameters for the second and all further runs.

Cost function and preprocessing

Practical control systems with several thousand lines of description, in which hardware-software partitioning is not obvious (and therefore cannot easily be done manually), may have a large design space. To reduce this design space, one could try employing knowledge of control system characteristics, component libraries, and synthesis tool or compiler properties. One way to do that is an approach known from synthesis as clustering with closeness criteria.[5] Closeness criteria are the use of common variables (data closeness), the probability of common execution of two operations (control closeness), and the similarity of operations that allow sharing of function units (operator closeness). Such global clustering, however, seems difficult here. The success of floating-point coprocessors, for example, would place a high weight on operator closeness. On the other hand, the common use of counters and timers in peripheral devices would place a high weight on data and control closeness.

We decided to use another approach, which we call hardware extraction.

Hardware extraction is the use of a partitioning cost function that favors for implementation in hardware those system parts that can be implemented well in hardware. Such a cost function encodes knowledge of synthesis, compilers, and libraries. Different cost functions can work in parallel or in sequence to extract different types of target hardware.

As an example, we developed a cost function to extract coprocessors for computation-time-intensive system parts, especially loops. As we have already mentioned, such coprocessors could be a valuable alternative to the library approach.

Simulation and profiling identify computation-time-intensive system parts. Given user-defined input patterns, we execute a simulation on the ES graph. We determine the number of times each node (including nodes representing subgraphs, such as function nodes) is executed. Next, we estimate the potential speedup through hardware synthesis and the communication penalty for nodes moved to hardware. Currently, we estimate on the basic-block level only (more precisely, we define a basic scheduling block that considers loops as basic blocks, too). We estimate the potential speedup with

- an operator table, which holds the execution times of the function units used in synthesis, and
- a local scheduling of the operations in the ES graph to estimate the potential concurrency, either using a simple list scheduling for a bounded number of hardware function units (currently a user-defined parameter) or using ASAP (as soon as possible) scheduling.

The estimate of communication time overhead of a basic block includes

- a dataflow analysis providing the number of variables to be communicated if this basic block alone is moved to hardware and the number of variables to be communicated if the adjacent blocks are moved to hardware as well; and
- the number of clock cycles for a variable transfer, given the processor type and communication mechanism.

All these are preprocessing steps and need not be repeated during inner-loop simulated annealing. ES graph basic blocks containing operations that cannot be mapped to hardware are excluded from this procedure.

We define costs incrementally. Currently, we partition only on the basic-block level. When a basic block B is moved to hardware, the cost increment dc is defined as

$$dc(B) = a(T_c, T_s) * [t_{\text{neff}}(B) + t_{\text{com}}(B) - t_{\text{HW-SW}}(B) - t_{\text{SW}}(B)] * lt(B)$$

where:

- $a(T_c,T_s) = \text{sign}(T_c - T_s) * \exp[(T_c - T_s)/T]$, 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 increasing the synthesis task by the unnecessary moving of basic blocks to hardware.
- $t_{\text{neff}}(B)$ is the effective hardware timing (synthesis result) for n function units, initialized with the local schedule mentioned earlier.
- $t_{\text{com}}(B)$, $t_{\text{HW-SW}}(B)$, and $t_{\text{SW}}(B)$ are the communication overhead, the hardware-software time overlap (in case of parallel execution; in the experiments $t_{\text{HW-SW}}(B) = 0$), and the runtime when the basic block is implemented in software, all initialized with estimated values.
- $lt(B)$ is the number of times the basic block was executed during profiling.

For the partitioning process we need knowledge of the hardware-software communication overhead. For m BSBs, $2^m - 1$ hardware partitions are possible; therefore, preprocessing of the exact communication costs is not practical. Instead, we estimate costs only for adjacent BSBs in the control flow. This avoids a global dataflow analysis.[15] Each BSB is attributed with a set (in_a) of variables used inside the block before they are defined and a set (out_a) of variables defined in a. These sets give an upper bound of the communication necessary when the BSB alone is moved to hardware.

Usually, we move several BSBs to hardware, and to avoid redundant variable exchange, we must consider communication between these BSBs. Let us consider BSB a as having been moved to hardware. The additional number of variable transfers from software to hardware is estimated as

$$\text{in}'_a = \text{in}_a - \bigcap_{b \in \text{predecessors}(a)} (\text{in}_b \cup \text{out}_b)$$

Estimations for out_a and for moving an operation back to software can be derived similarly.

To keep the coprocessor overhead small, we use a small, fixed shared-memory space. If variable var i will be transmitted to a coprocessor register, it is first moved to the shared memory with load and store operations. Assuming fixed-size data values, the time $t_{\text{com}}(a)$ is proportional to the number of elements in the in sets and out sets. When estimating $t_{\text{com}}(a)$, we must take memory and register allocation into account. Figure 4 outlines the communication steps for a single variable transfer through shared memory.

In our example, each variable communication requires eight clock cycles corresponding to up to eight instruction executions on the processor. This means that communication overhead minimization is important in fine-grain

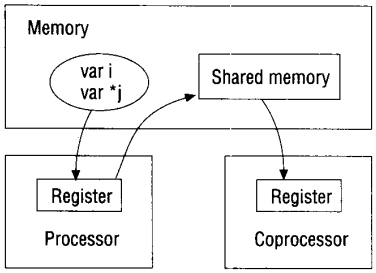


Figure 4. *Variable communication path.*

hardware-software partitioning. Therefore, the translation from the ES graph to HardwareC uses a more precise communication analysis than an analysis of adjacent blocks.[16] In case of arrays, only pointers are communicated via the shared memory (Figure 4, var *j).

The cost function does not explicitly account for hardware costs. Instead, for our experiments, the user provides the number *n* of functional units in the coprocessor as a hardware cost parameter, and the system optimizes the timing. In future experiments, we will add hardware costs to the cost function, as provided by the synthesis tool.

Target architecture and communication protocol

The experimental results presented in the next section are based on the target architecture shown in Figure 5. The standard Sparc processor[17] communicates with a synthesized coprocessor via memory communication. At present, software and hardware execute in mutual exclusion, and the Sparc and the coprocessor are coupled by the principle of communicating sequential processes. We are also working on other communication mechanisms.

When the processor writes to a predefined reference address, a Start signal is issued to the coprocessor and a Hold signal to the processor (BHold: Hold signal, AOE: address bus enable, DOE: data bus enable). The data word written to the reference address is the BSB-Id (Figure 3), which indicates the hardware function to be executed. The Sparc switches to the Hold state and the coprocessor enables its data and address lines. The coprocessor decodes the BSB-Id and begins executing the corresponding code segment. At the end of the segment, the Done signal is issued, the coprocessor stops, and the Sparc processor leaves the Hold state.

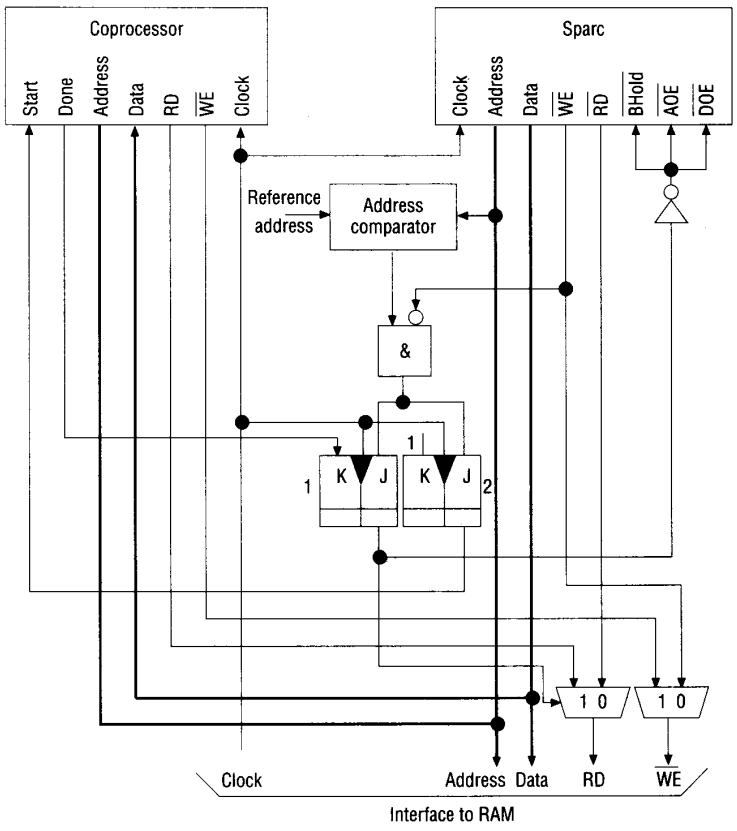


Figure 5. *Memory-coupling circuit.*

Experiments

For demonstration we start with a manually partitioned example because of the simplicity of the result. The example is a practical algorithm, a chroma-key algorithm for high-definition television studio equipment.[18] The desired response time was 1 second. The algorithm needed 3 seconds on a Sparc1+. The program has 1,400 lines of C code. In the manual experiment, we applied a simplified cost function to partition two loops with 34 lines of code, which are iterated 10,070 times, taking 90% of the computation time. Figure 6 (next page) shows the program section with the two loops, 30c and 30d, shaded. This hardware partition consists of several consecutive BSBs, but it is still fine-grain compared to function- or task-level partitioning. The coprocessor executes the two BSBs (30c and 30d) and is therefore called 10,070 times in each process execution.

We translated the section to HardwareC. The variables *cr*, *cr*1, *cr*2, and *cb* are

```
while (cb <= cb2 + key1) {
    if (cb > vtab[cr] ) {
        if (cb >= htab[cr] )
            kt[cr] [cb] = 255 ;
        else {
            iabsv = 512 ;  /* = 256 + 256 */

/* ================ area 30c ================ */
                for (i = cr1; i <= cr2; i++) {
                    ihilf = 1abs (cr - i) + 1abs (cb - vtab [i] ) ;
                    if      (ihilf < iabsv) {
                            iabsv = ihilf;
                    }
                } /*of for i*/
                iabsh = 512 ;  /* = 256 + 256 */

/* ================ area 30d ================ */
                for (i = cr1; i <= cr2; i++) {
                    ihilf = 1abs (cr - i) + 1abs (cb - htab [i] ) ;
                    if      (ihilf < iabsh) {
                            iabsh = ihilf;
                    }
                } /*of for i*/

/* ================ area t1 ================ */
                kt[cr] [cb] = iabsv * 255 / (iabsv + iabsh) ;
            }     /*of else*/
        }
        FORLIM = min (cr + keyr, cr2) ;

        for (v = max(cr1, cr - key1); v <= FORLIM; v++) {
            FORLIM1 = min (cb + keyr, cb2) ;

            for (u = max(cb1, cr - key1); u <= FORLIM; u++) {
            kt[v] [u] = kt[cr] [cb] ;
            }
    }

    cb += keyf;
}  /*of while cb <= cb2*/
```

Figure 6. *Loops in chroma-key algorithm.*

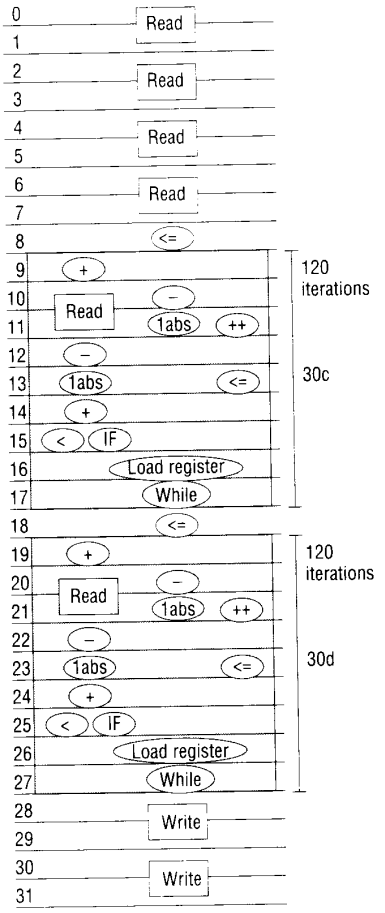


Figure 7. *Scheduled loops.*

read from memory upon activation of the coprocessor, and the values *iabsv* and *iabsh* are written to memory immediately before control returns to the processor. The table values in *vtab* are read during processing. Olympus provided the schedule in Figure 7, which uses two ALUs.

The result was a circuit with 17,300 gate equivalents and a 120-ns clock cycle time using the LSI 1.5-μm library. After manually inserting a few drivers into high-fan-out nets, we reduced the coprocessor cycle time to the processor cycle time of 30 ns with 18,000 gate equivalents. The loop execution time was 65.2 μs per coprocessor call and 0.65 sec. for all 10,070 calls. The total execution time was 1 sec., a speedup of 3, at much less cost than a second Sparc core.

More interesting for synthesis is automatic partitioning. We obtained the results for our next examples with a *fully automatic* partitioning process, excluding the outer loop:

1. System specifications
2. Translation of system specifications to the ES graph representation by the C^x compiler

3. Partitioning by simulated annealing
4. Mapping to software and hardware descriptions by translating tools
5. High-level synthesis and software compilation
6. Runtime analysis

Except for the Olympus system, all tools belong to the Cosyma system.

We selected benchmarks to demonstrate the feasibility of an automatic partitioning process, although its efficiency is not yet optimal. Much work is still needed to exploit the full optimization potential. In the examples, T_c is moderately defined as $T_c = 1/2\ T_s$ (T_s for an all-software solution), so that the HardwareC description is small enough that Olympus can finish within a couple of hours on a Sparc10/41. In all cases, the maximum number n of function units to be implemented in hardware was limited to 1. A function unit is assembled from a 32-bit ALU, which is built from bit-slice components (Texas Instruments 74×181), and a 32-bit nonpipelined multiplier that needs two clock cycles for a multiplication.

Our results, shown in Figures 8 and 9, demonstrate the behavior of simulated annealing for some realistic benchmarks, Simp and Fft. We executed the partitioning 10 times for each benchmark. Each partitioning run started with a different random seed. Two interesting aspects are the selection of the basic blocks and the repeatability of the results for different initial conditions (random seeds) of simulated annealing, suggesting the usefulness of the cost function. Figures 8a and 9a show the number of times each basic block was moved to hardware; Figures 8b and 9b show the number of iterations of a block during profiling. As already mentioned, the desired speedup was set to 2.

Figure 8 shows that two of the three most often extracted blocks correspond to computation-intensive code segments. One of them was extracted in all 10 runs. Here the factor *It* of the cost function

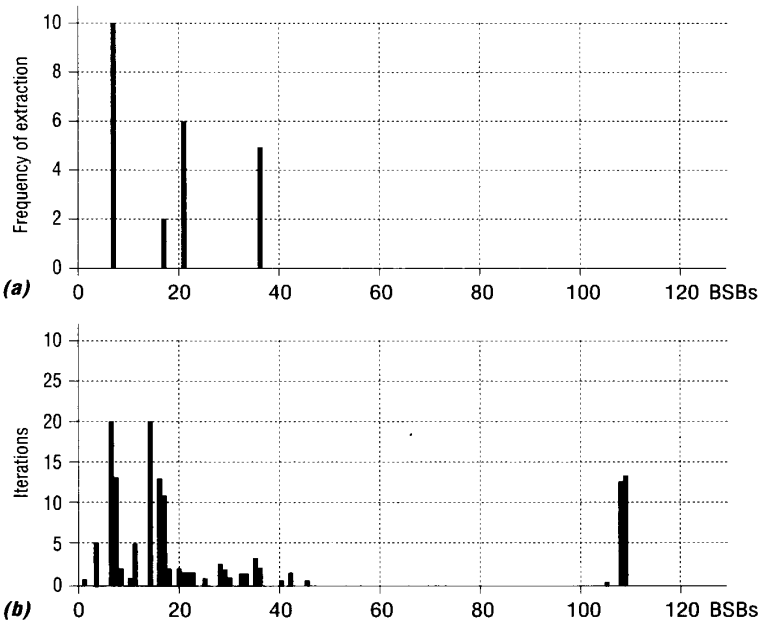


Figure 8. *The* Simp *benchmark: probability of extraction* ***(a)*** *and frequency of iteration* ***(b)****.*

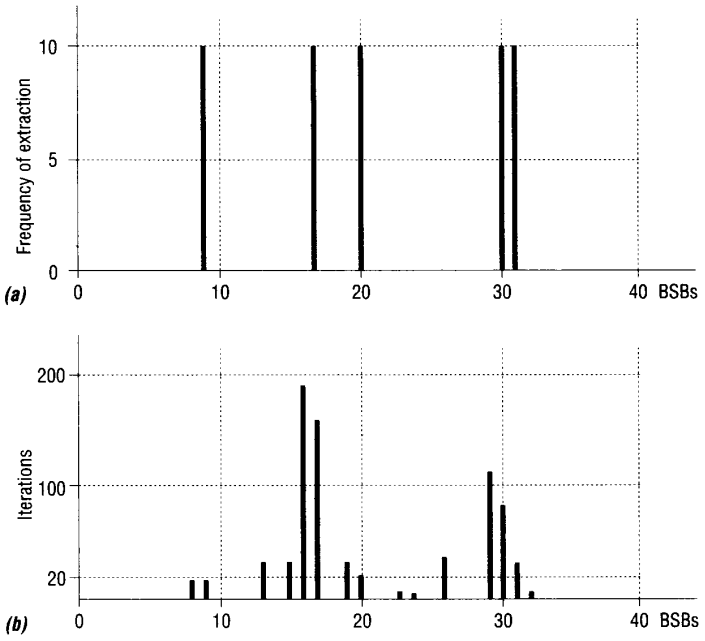


Figure 9. *The* Fft *benchmark: probability of extraction* ***(a)*** *and frequency of iteration* ***(b)****.*

Table 1. Partitioned benchmarks.

Benchmark	Clock cycles used SW	HW-SW	t_c (%)	Speedup
Diesel	22,403	16,394	9.9	1.4
Smooth	1,781,712	1,393,525	49.6	1.3
3d	1,377	1,514	13.8	0.9

dominates. At first glance, it is surprising that the two blocks 108 and 109 with peak execution rates in Figure 8b never move to hardware. A closer look at these code segments shows that the speedup of coprocessor hardware would not have been large enough to compensate for the communication overhead t_{com}. This negative gain is multiplied by the execution rate, making an implementation in hardware very unlikely.

For the benchmark shown in Figure 9, again most extracted basic blocks correspond to computation-intensive code segments. Obviously, all five blocks are extracted in all 10 partitioning runs, reflecting that the blocks are iterated relatively often and that communication overhead is small enough relative to operation speedup. As other experiments showed, these benchmarks are exemplary. In almost all cases, they could reach the required speedup, supporting our approach of driving the annealing process toward a feasible design point using an exponential cost function.

We experimented with three more benchmarks. The results assume that hardware and software use the same instruction cycle (as in the HDTV example). We are not at all limited to this assumption, but it simplifies comparison of system performance before and after partitioning.

The Diesel benchmark (see Table 1) is a real-time algorithm for the digital control of a turbocharged diesel engine. Our timing analysis tool calculated a computation time of T_s = 22,403 cycles on a Sparc1+ processor. An automatically generated hardware-software codesign could reach a speedup of 1.4 (16,394 cycles) under the given constraints.

The second benchmark, Smooth, executing a filter algorithm on a digital image, gave almost the same result.

The third benchmark, 3d, is an interesting example showing that automated hardware-software codesign is not a trivial task. Under the same conditions where Diesel and Smooth resulted in a real speedup, we achieved a "speedup" of 0.9 with *3d*. Here the partitioning did not consider the optimizing potential of the GNU C-compiler. An investigation showed that a code segment consisting mainly of two multiplications was extracted to hardware. The partitioning algorithm assumed two cycles per multiplication. Including transfer times, a hardware realization would amount to less than 10 cycles, and software execution would take about 40 cycles (in our experiments, we took the estimated execution times of the Sparc processor from a table). Studying the assembler code produced by the GNU C-compiler revealed, however, that both of the multiplications had one constant factor converted to additions and shifts.

Table 1 also shows the communication overhead for the three benchmarks. An amount of up to 50% is not unusual for the group of benchmarks (real-time applications for small embedded systems) we investigated. Reducing communication overhead seems to be one of the most important factors in gaining a higher speedup.

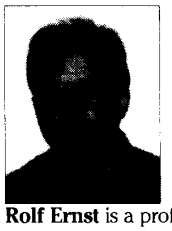
For a higher speedup we also need a synthesis tool that schedules and optimizes across basic-block boundaries, using some of the techniques mentioned earlier. We are currently working on such an approach.[19]

FINE-GRAIN HARDWARE-SOFTWARE partitioning is feasible and useful in microcontroller design. The results of automated software-oriented partitioning with hardware extraction are promising and are similar for different initial conditions. The examples and the partitions we have presented are not trivial. We expect further improvements from optimized communication mechanisms, and, in particular, from synthesis with high-level transformations, pipelining, and scheduling across basic blocks. Precise estimations of synthesis results and shorter synthesis computation times are very important for the industrial use of cosynthesis systems such as Cosyma. D&T

References

1. M.B. Srivastava and R.W. Brodersen, "Rapid-Prototyping of Hardware and Software in a Unified Framework," *Proc. Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 152-155.
2. K. Buchenrieder and C. Veith, "CODES: A Practical Concurrent Design Environment," handout from Int'l Workshop on Hardware-Software Codesign, Estes Park, Colo., Oct. 1992.
3. P. Windirsch et al., "Application-Specific Microelectronics for Mechatronic Systems," *Proc. Third European Design Automation Conf.*, IEEE CS Press, 1992, pp. 194-199.
4. E. Barros and W. Rosenstiel, "A Method for Hardware/Software Partitioning," *Proc. Compeuro*, IEEE CS Press, 1992.
5. E. Dirkes Lagnese and D.E. Thomas, "Architectural Partitioning for System-Level Design," *Proc. 26th Design Automation Conf.*, IEEE CS Press, 1989, pp. 62-67.

6. P. Athanas and H.F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *Computer*, Vol. 26, No. 3, Mar. 1993, pp. 11-18.
7. R.K. Gupta and G. De Micheli, "System-Level Synthesis Using Re-programmable Components," *Proc. Third European Conf. Design Automation*, IEEE CS Press, 1992, pp. 2-7.
8. R.K. Gupta, C.N. Coelho, Jr., and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," *Proc. 29th Design Automation Conf.*, IEEE CS Press, 1992, pp. 225-234.
9. N. Woo, W. Wolf, and A. Dunlop, "Compilation of a Single Specification into Hardware," handout from Int'l Workshop Hardware-Software Codesign, Estes Park, Colo., Oct. 1992.
10. B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Trans. Software Engineering*, Jan. 1985, pp. 80-86.
11. G. De Micheli et al., "The Olympus Synthesis System," *IEEE Design & Test of Computers*, Vol. 7, No. 5, Oct. 1990, pp. 37-53.
12. T. Benner, J. Henkel, and R. Ernst, "Internal Representation of Embedded Hardware/Software Systems," to be published in *Proc. Second IFIP Int'l Workshop Hardware/Software Codesign*, IFIP, Geneva, 1993.
13. W. Ye et al., "Fast Timing Analysis for Hardware-Software Cosynthesis," *Proc. Int'l Conf. Computer Design*, IEEE CS Press, 1993, pp. 452-457.
14. R. Potasman, "Percolation-Based Synthesis," *Proc. 27th Design Automation Conf.*, IEEE CS Press, 1990, pp. 444-449.
15. A.W. Aho, R. Sethi, and J.D. Ullmann, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., 1986.
16. G. Glawe, *Erstellen eines Code-Generators zur Umsetzung eines C-Syntax-Graphen in die Hardwarebschreibungssprache HardwareC [A Code Generator for the Translation of a C Syntax Graph into the Hardware Description Language HardwareC]*, master's thesis, Technical Univ. of Braunschweig, Germany, May 1993.
17. Cypress Semiconductor, *Data Book*, Cypress Semiconductor, San Jose, Calif., 1992.
18. C. Ricken, *Optimierung der automatischen Einpegelung eines HDTV-Chromakey-Mischers [Optimization of an Automatic Color Level Control for an HDTV Chroma-key Blue Screen System]*, master's thesis, Technical Univ. of Braunschweig, 1992.
19. U. Holtmann and R. Ernst, "Experiments with Low-Level Speculative Computation Based on Multiple Branch Prediction," *IEEE Trans. VLSI Systems*, Sept. 1993.



Rolf Ernst is a professor of electrical engineering at the Technical University of Braunschweig, Germany. His research interests are VLSI CAD and digital circuit design. Previously, he was a member of the technical staff in the CAD and Test Laboratory of AT&T Bell Laboratories and a research assistant at the University of Erlangen, Germany. He holds a diploma in computer science and a PhD in electrical engineering from the University of Erlangen. He is a member of the IEEE, the IEEE Computer Society, and the German GI (Society for Computer Science).

Jörg Henkel is pursuing a PhD in electrical engineering at the Technical University of Braunschweig, where he is involved in the development of Cosyma, the experimental system for hardware-software codesign. In addition, his interests include high-level synthesis and computer architecture. He received a diploma in electrical engineering from the Technical University of Braunschweig.

Thomas Benner is a research assistant at the Technical University of Braunschweig, working on the Cosyma project. His interests are hardware-software codesign, computer architecture, graph theory, and digital circuit design. He holds a diploma in computer science from the Technical University of Braunschweig, where he is pursuing a PhD in electrical engineering. He is a member of the German GI (Society for Computer Science).

Address correspondence to Rolf Ernst, Institut für Datenverarbeitungsanlagen, Technische Universität Braunschweig, Hans-Sommer-Str. 66, D-38106 Braunschweig, Germany; ernst@ida.ing.tu-bs.de.