Hardware generation and partitioning effects in the COSYMA system

J. Henkel, Th. Benner, R. Ernst
Technische Universität Braunschweig
GERMANY
Email: henkel@ida.ing.tu–bs.de

Abstract

This paper reports on the current results that have been gained by a fully automatically hardware/software partitioning process in our co–synthesis system COSYMA [ErHe92]. By means of some examples we will describe the role of the hardware/software communication overhead and the way the partitioning procedure is influenced by communication.

1 Introduction

The range of applications for embedded real-time systems is steadily increasing: car electronics, office automation, telecommunication etc., benefit from progress in embedded system performance with the consequence of more functionality and complexity at lower prices.
Hardware/software co-design plays 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, optimization goals like area minimization, low power consumption etc. and time constraints, formulated in a high level description. In the future the designer will mainly concentrate on the specification – the hardware/software tradeoff is determined by an interactive or automatic partitioning procedure, and descriptions for both, hardware and software part, are generated. This will help to minimize the turnaround times can be minimized.
The architecture of a real-time system depends on parameters like scheduling algorithms, processor architecture, communication concepts, etc. A base for starting hardware/software co-design is an important class of embedded systems – single chip micro-controllers. They offer a wide field of functionality and applications in order to test partitioning algorithms, optimization goals, etc. A loss in optimality compared to ASICs is compensated by much shorter design times and flexibility.
Recent trends support the investigations in hardware/software co-design: todays micro-controllers are increasingly programmed above the assembly language level, mostly using C. That makes the partitioning algorithms independent of the particular hardware, and 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 concerning an automatic hardware/software partitioning that is – in our opinion – one of the key issues in hardware software co-design. The VULCAN system [GuMi92] and the COSYMA system [ErHe92] perform automatic partitioning under time constraints and optimization goals. Other systems like [WoWoDu92] also approach this aim.
Both VULCAN and COSYMA have the same target architecture, small embedded architectures, such as micro-controllers, consisting of a predefined processor core and synthesized or user defined co-processors, memory and peripheral units. Both use a partitioning approach which iterates over hardware synthesis and software generation. Iteration seems to be necessary, because there are no approaches known to accurately estimate the results of optimizing compilers and high-level synthesis tools with advanced techniques, such

as loop folding, tree height reduction or percolation based synthesis [Po90].
While VULCAN is hardware oriented, starting with an all hardware solution (except for data dependent loops) and moves operations to software on a given processor kernel until time constraints are violated, COSYMA is software oriented starting with an all software solution on a given processor kernel and moves operations to peripheral hardware until no time constraint is violated any more. An advantage of starting with an all software solution is that no restrictions on the system input description are necessary (dynamic data structures, pointers,...)[1]. It still requires a somewhat more complex internal data structure and partitioning. Both approaches assume implicitly that an application specific hardware executes faster than an equivalent software solution.

This paper is structured as follows: the next section describes in short terms the tools in the COSYMA system and the principle of partitioning. The section 3 focusses on problems in translating the internal graph representation ESG into a hardware description, and describes the communication mechanism we currently use. Then the section 4 shows some automatically gained partitioning results and section 5 discusses aspects to be improved in order to gain a greater speedup. Finally section 6 gives a conclusion.

2 The COSYMA System

Fig. 1 gives an overview on the system **COSYMA** (**CO**–**SY**nthesis for e**M**bedded **A**rchitectures). COSYMA accepts a superset of C as an input. This superset, C^x, contains time constraints, multitasking and process communication. To control the partitioning process, the user can predefine functions to be mapped to hardware and functions to be implemented in software in order to keep them flexible. C^x was pragmatically defined to have access to C programs as benchmarks.
A C^x compiler translates the input description into an extended syntax graph, the ES graph [BeHeEr93]. The ES graph permits partitioning at the statement, basic block and function levels. Some more details on the graph will be given later.
There is a simulator for the ES graph which allows simulation of a C^x description (with parallel processes) and profiling. The profiling information is required for cost function computation and timing analysis.
Parts of the ES graph which will 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, while data to be exchanged are derived from a data flow analysis.
Because of the similarity of C^x and C and because the syntactical structure is maintained in the ES graph, 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 [Cyp89].
Then, a run time analysis checks if the timing constraints given in C^x are met. We found that, to be accurate enough, complex processor cores with load–store architecture, pipelining and concurrent units require an analysis which models the data and control flow in the processor [YErBeHe93][Ha92], which known approaches to run time analysis (e.g. [PaSh91], [PuKo89]) cannot provide. Also, hardware timing must be included. Experiments with a larger example, however, lead to RT–level simulation times of more than 1/2 hrs on a SPARC 10/41 for each input pattern, even in the un-partitioned case. Iteration would be impractical under these circumstances. So, we developed a hybrid approach (in the following named HTA tool) combining RT–level simulation and formal analysis.
The ES graph parts which will be implemented in hardware, i.e. coprocessor or peripheral hardware, are mapped to the hardware description language of the high–level synthesis system. The communication protocol is, again realized using templates with input and output statements. Currently, we use the the high–level

[1] This, of course, does not mean that all of these semantic constructs can be moved to hardware.

synthesis system OLYMPUS [MiKuMa90] from Stanford, which uses HardwareC as input language.
Because OLYMPUS synthesis and run time analysis are computation time intensive, the iterative partitioning uses a cost function which is based on estimated hardware cost and timing data. As a first approach, the partitioning procedure is defined as the following optimization problem: given an user defined amount of hardware components (i.e. ALU's, multiplier) the task is to find a minimum set of code segments that lead to a sufficient speedup when implemented in hardware. As described in [HBYSEG93] we use the simulated annealing heuristic because of

- the flexibility in cost function,
- the possibility of a *quality/computation time* trade off,
- and a mathematically well investigated behavior

The partitioning goals are in order of decreasing importance:

- meet real-time constraints
- minimize hardware costs, currently determined by the given number of units
- minimize computation time for a design and, thus, the turnaround time

While the first goal is a *hard* constraint (because it is a necessary condition), the other ones are *soft* optimization goals. Here a special treatment is used to achieve compliance to the hard constraints [HBYSEG93]. This is what we call the *inner partitioning loop.* In fig. 1, it is marked with a dotted surrounding box.

The outer loop including synthesis and run time analysis provides hardware cost and timing data. The actual timing and costs are then used to adapt the estimated values of the partitioning cost function of the inner loop. While 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 time and must be limited in size.

3 Generating a hardware description from internal representation

Target architecture

The target architecture for COSYMA [ErHe92] is a single processor (currently SPARC), one or more coprocessors[2] and peripherals[3]. Communication between software and coprocessor is done by memory coupling and a single CSP type protocol (Communicating Sequential Processes). Both sides execute in mutual exclusion. We use this very simple protocol, but we are going to examine other mechanisms to allow parallel execution in the processor and the coprocessor.
The hardware description derived from the partitioned graph and the communication are to a large degree tool and architecture dependent. We will highlight some details to give an impression of the problems and our approach.
When several segments of the C^x language are moved to hardware, they can be mapped to different function of the same coprocessor. Therefore a mapping procedure is required, which assigns each call from the software to the corresponding hardware function. The HardwareC description is structured as a "case"-statement. An index is transferred, whenever the coprocessor is called by the software, indicating which function is to be executed.

[2] In this connection we use the term *coprocessor* only for an application specific hardware, partitioning by means of standard coprocessors is not yet supported but under development.

[3] Peripheral devices are currently user supplied

Communication estimation

The elements of an ES graph are generated as the result of C^x compilation. The ESG does not yet contain explicit information required to estimate hardware execution times and determine communication costs. Therefore, the syntax graph is overlayed by a second graph consisting of cross-linked basic blocks, in the following called BSB (*basic scheduling blocks*). These BSBs contain local DFGs (data flow graphs) on which the local scheduling is executed.

Each BSB is attributed with a set (IN_s) of variables which are used inside the block before they are defined and a set of variables OUT_a, that are defined in a. These sets give an upper bound of the communication necessary when the BSB alone is moved to hardware.

When several BSBs are moved to hardware, communication between these BSBs has to be regarded to avoid redundant variable exchange. In order to avoid a time consuming global data flow analysis, the analysis is restricted to BSBs adjacent in the control flow. Let BSB a be considered to be moved to hardware. Then, the additional number of variable loads is estimated as:

$$IN'_a = IN_a - \bigcap_{b \in predecessors(a)} (in_b \cup out_b). \tag{1}$$

Estimations for out_a and moving an operation back to software can similarly be derived. Currently, the cost function calculation limits the partitioning to basic block granularity.

Figure 3 shows a program segment represented by BSBs. On the left there are those ones which are implemented in software and on the right side those which are implemented in hardware. Here it is assumed that BSB a and BSB b have already been moved to hardware (in a earlier phase of the partitioning algorithm) and BSB c will now be moved to hardware. The upper ellipse in each BSB contains the set IN_s and the lower one the set OUT_a. In the example BSB c uses the input variable d that is also a output variable of a predecessor (BSB b). That means this variable has not to be transferred from software to hardware (no additional communication). In many cases the communication cost is lowered when moving more sequential BSBs to hardware.

Assuming fixed size data values, the communication time t_c is nearly proportional (we will discuss this in section 5) to the number of elements in the *in*–sets and *out*–sets. When estimating t_c, memory and register allocation must be taken into account. Fig. 2 outlines the communication steps for a single variable transfer through shared memory. In order to keep the co–processor overhead small, we use a small and fixed shared memory space. If variable *var i* shall be transmitted to a co–processor register, it is first moved to the shared memory with a load and a store operation. In our example, each variable communication requires 8 clock cycles corresponding to up to 8 instruction executions on the processor. So, the communication overhead minimization is important in fine grain hardware-software partitioning.

Communication implementation

The communication procedure is very simple and is not yet very efficient. Scalar values and pointers to vectors are written into a shared memory area. Whenever the coprocessor is started, it reads all parameters and global variables, which are needed. The objects which have to be transferred are derived from the ESG by a dataflow analysis, not only limited to adjacent blocks as for estimation. Arrays have to be located in the shared memory area, too, but only their base addresses are read by the coprocessor at the start of the execution. After computation of the hardware function, the results and global variables are written back to the shared memory area. An example of a coprocessor description is outlined in the following:

```
index=receive_function_index();

switch(index) {                           /* switch on the function */
```

```
                                        /* index                 */
    case FUNC_1 :
            read_parameters();          /* communication SW->HW   */
            read_global_variables(STARTADDRESS);
            compute_result(STARTADDRESS+8);
            write_result(STARTADDRESS);/* communication HW->SW   */
            write_global_variables(STARTADDRESS+4);
            break;
     case FUNC_2 :  ...
            break;
}
```

The corresponding software writes the global variables and parameters into the shared memory area and calls the coprocessor. The addresses of the variables are fixed and known by the software and hardware. The coprocessor is started by writing the hardware function number to a special address. After the execution of the hardware the global variables are updated and the the results are read from the common memory. Currently, we are only able to handle 32 bit values, but we are going to extend the mechanism to data types with another length. The corresponding software template is shown in the following:

```
/* communication SW->HW                */
/* write global variables and parameters */

*(STARTADDRESS)   =    globvar1;
*(STARTADDRESS+4) =    globvar2;
*(STARTADDRESS+8) =  param1;

/* call the coprocessor               */
*(COPROSTART) = co_function_index;

/* communication HW->SW               */
result   =  *(STARTADDRESS);
globvar1 =  *(STARTADDRESS+4);
globvar2 =  *(STARTADDRESS+8);

...
```

The communication mechanism presented above allows only the call of the coprocessor from the software. Conversely the call of the software from the hardware is not supported. As a consequence, every function which is called by a hardware function directly and indirectly has to be extracted, too. This could be avoided through an extension of our currently used communication mechanism, in order to enable a call of the software from the hardware.

4 Partitioning results

The automated partitioning procedure is the main goal of our software-oriented approach for hardware/software co-design. Details, especially concerning the optimization algorithm, are discussed in [HBYSEG93] or [ErHeBe93].

Benchmark	used clock cycles		t_c [%]	speedup
	SW [cyc]	HW/SW [cyc]		
diesel	22 403	16 394	9.9	$\simeq 1.4$
smooth	1 781 712	1 393 525	49.6	$\simeq 1.3$
3d	1377	1514	13.8	$\simeq 0.9$

Table 1: Partitioned benchmarks

The presented results are obtained by the COSYMA system tools without any manual support concerning the inner loop partitioning. Hardware synthesis has been done by the OLYMPUS synthesis system [MiKuMa90]. The benchmarks shall demonstrate the feasibility of an *automatic* optimizing partitioning process, while the reached efficiency is not yet optimal.
The following results assume that hardware and software parts are executed using the same instruction cycle. We are not at all limited to this assumption but it simplifies the comparison of the execution times (before and after partitioning). In order to keep synthesizing and (hardware-) simulation times low we restricted the tool to design points aiming at a speedup of ≤ 2 with one ALU and one multiplier (a 32 Bit ALU that consists of 74x181 ALU's and one multiplier requiring two instruction cycles for a 32 Bit multiplication) were no chaining or pipelining is used).
First, we took the benchmark named *diesel* (see table 1), a real-time algorithm for the digital control of a turbocharged diesel engine[MoYo87]. Our timing analysis tool HTA determined a computation time of 22.403 cycles when running on a SPARC1+ processor. An automatically generated hardware software co-design could reach a speedup of 1.4 (16394 cycles).
Another benchmark, *smooth* executing a filter algorithm on a digital image gave almost the same result.
An interesting example in order to show that automated hardware/software co-design is not a trivial task is demonstrated by means of the third benchmark. With the same conditions where *diesel* and *smooth* resulted in a real speedup we got a "speedup" of 0.9. Effects that are responsible for such a result are discussed in the next section.

5 Problems in iterative partitioning

At the current time we gain a speedup of 1.4 by means of a fully automated partitioning process. As hand-supported and hand-optimized experiments showed [ErHe92], a speedup of 3 is possible.
Here we will discuss the discrepancy between the desired speedup the designer requires from the COSYMA system and the actual speedup that is gained. At the moment we get a speedup of only 1.4 when 2.0 was the parameter sent to COSYMA.

As explained in section 2 a schedule for every BSB is made in order to give an early estimation value for an expected speedup, assumed that a BSB is transferred to hardware. Table 2 shows the real execution times of four real-time benchmarks (obtained by the HTA tool; the units are clock cycles) when running on a SPARC1+ processor. Columns 2 and 4 contain the according cycles of the assembly code with (*optim* i.e. -*O2* flag of the GNU C compiler) and without (*no optim*) optimization flag at compile time. Column 6 gives the estimated execution times gained by a simple but very fast table look-up: each operator in the data flow graph is represented by one or more processor instructions. For each instruction a constant execution time is assumed and so the execution time of a whole BSB and finally of the whole benchmark has been calculated, also using profiling information. Hereby data dependencies and consequently pipeline interlocks are neglected.

As seen, in all cases the estimated execution (in cycle times) is greater than the optimized real execution

Benchmark	Simulation				look-up
	no optim. [cyc]	dev.[%]	optim. [cyc]	dev.[%]	
diesel	31 279	-27.3	22 403	1.6	22 752
fft	40 665	-37.9	16 350	54.5	25 262
smooth	5 158 932	-54.0	1 781 712	32.7	2 265 247
3d	4 586	-33.6	1 377	121.0	3 043

Table 2: Comparison of simulation and look-up table estimation

time and less than the non-optimized one. In general the deviations are too large in order to aim at a better relation between a desired speedup and a gained speedup. Therefore we will use the HTA tool for calculating the software execution times of single BSB's which are candidates for implementation in hardware.

A particularly bad case is shown in table 2 were are seen the real execution times (obtained by our HTA tool [YErBeHe93]) of three real time benchmarks before partitioning (all-software solution) and after partitioning (hardware/software co-design are shown). The partitioning procedure of the *3d*-benchmark resulted in a "speedup" of 0.9. An investigation revealed that the communication that is included in the figures dominated the small speedup of the code segments that were transferred to hardware. Among other operators, two multiplications which were assumed to need about 28 cycles (on a SPARC1+) were hidden in these code segments. The hardware execution time (t_{HW}) would result in about two cycles per multiplication and about 20 cycles for the communication[4] ($t_{HW} = 24\ cycles$). In reality the compiler converted one of the multiplications to a *shift* instruction (1 cycle) and the other multiplication to an *add* instruction (an iterated multiplication within a loop with a multiplicand that itself was incremented by a constant value; together: $t'_{SW} = 2\ cycles$). This effect caused a "speedup" of 0.9.
Here it seems to be a better idea to base the estimation on the HTA-tool instead of a statement table look-up. This is currently under implementation. Here, again, the necessity of a precise timing analysis tool in automatic hardware/software partitioning is shown. An important overhead factor is the relatively large communication time in table 1 (t_c).
At the current time our communication protocol (see section 3) is very simple and run-time intensive. We will discuss an example. Assuming a BSB would be moved from software to hardware and the communication between hardware and software parts is done by means of the CSP principle, the time for communication amounts to:

$$t_c = m\ (t_{SW\_w} + t_{HW\_r}) + n\ (t_{HW\_w} + t_{SW\_r}) + t_{switch} + q\ (t_{SW\_r} + t_{SW\_w}) \tag{2}$$

where:

m is the number of input values of that code segment (BSB), IN_b

n is the number of output values of that code segment (BSB), OUT_b

q is a factor that takes into account the number of variables which are not stored in the register set of the processor (compiler dependent). In our estimation we set $q = 0$.

t_{SW_w} is the time needed to write data values to the shared memory segment from the software side

t_{HW_r} is the time needed to read data values from the shared memory segment from the hardware side

t_{HW_w} is the time needed to write data values to the shared memory segment from the hardware side

[4] communication time calculation is explained later.

t_{SW_r} is the time needed to read data values from the shared memory segment from the software side

t_{switch} is the time to switch from software to hardware and vice versa

In case of array accesses we only transfer the pointer of the first element. With our SPARC processor implementation [Cyp89] we get:

$$t_{SW\_w} = 3\ cycles\ (store\ instruction) \tag{3}$$

$$t_{HW\_r} = 2\ cycles \tag{4}$$

On hardware side:

$$t_{HW\_w} = 2\ cycles \tag{5}$$

$$t_{SW\_w} = 2\ cycles\ (load\ instruction) \tag{6}$$

In the example in figure 4 m is 3 and n is 2, so we get for t_c with $t_{switch} = 2\ cycles$:

$$t_c = 3(3+2) + 2(2+2) + 2 = 25\ cycles \tag{7}$$

Consequently, in most cases the communication costs prohibit an extraction of small code segments in order to avoid a "speedup" of < 1.

Another example demonstrates the influence of the communication overhead in the partitioning process. Table 3 shows two code-segments (BSB a and BSB b) of a benchmark and their statistics, where

iterat is the number of times that BSB was executed during a simulation run,

t_{HW} is the number of cycles a coprocessor would need to execute this BSB (estimation value, as explained above),

t_{SW} is the number cycles the SPARC processor would take to execute this BSB (estimation value, as explained above),

sp' is the speedup gained when executing in hardware instead of software, not taking into account the communication overhead

t_c is the communication time in cycles

sp is the speedup taking into account the communication

In this example BSB a has been moved to hardware and BSB b remained in software. Furthermore a speedup of 2.0 has been required from the partitioning tool. Interesting is the fact that sp' of BSB b amounts to 14.5 whereas sp' of BSB a only amounts to 10.6. The following code segment shows BSB a and BSB b: where BSB a needs only 7 variables or addresses (*kernel_cols, sumval, IN, row, i, column, FIL*) to be transferred vom software side at an amount of 10 operations, BSB b would need 3 variable transfers at an amount of only one operation. *With* communication it results in a gained speedup of 2.46 for BSB a and only 1.81 for BSB b.

BSB	iterat	t_{HW} [cyc]	t_{SW} [cyc]	sp'	t_c [cyc]	sp
a	34 596	207 576	2 214 144	10.6	691 920	2.46
b	3 844	7 688	111 476	14.5	53816	1.81

Table 3: Statistics of a partitioned and an un–partitioned BSB

```
    .
    .
    .
for(column = 0; column < ('\x40' - kernel_cols) + '\x1'; column++){

   sumval = 0;
   for(i = 0; i < kernel_rows; i++){

      for(j = 0; j < kernel_cols; j++){                         /* This is */
         sumval += IN[(row+i)*256+(column+j)]*FIL[i*'\x3'+j];  /* BSB a   */
      }

   }
      .
      .
      .
   c2 = sumval/normal_factor;  /* This BSB b */
      .
      .
      .

}
    .
    .
    .
```

This again demonstrates the importance of an efficient communication protocol that is suited to a specific application. We are currently working on improving the communication.

6 Conclusion

We have presented some experience with our COSYMA system for the automated co-synthesis of hardware/software systems.

The experiments showed that the partitioning process can be executed fully automatically. The results are not yet optimal, meaning that the gained speedup is too low. As shown, above all other things the communication overhead is responsible here. We explained the effects. One focus of our current and future is implementing and developing communication mechanisms which are suited to a specific application. Furthermore the estimation techniques have to be improved. The potential for improvements has already been demonstrated in [ErHe92] and [HoEr93].

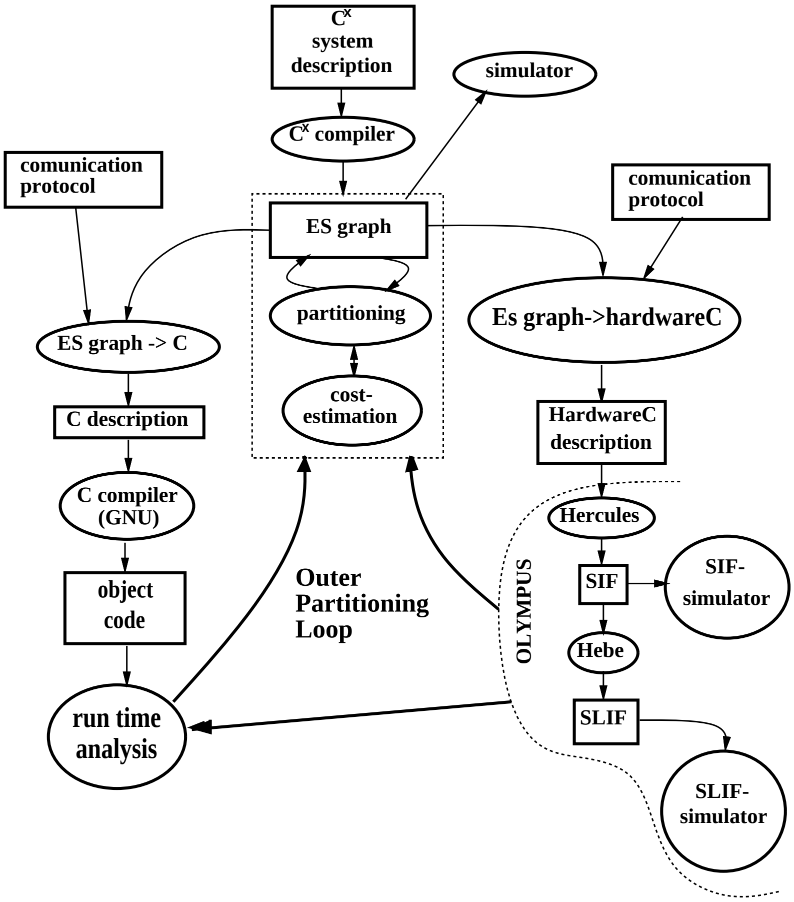

Figure 1: The COSYMA experimental system

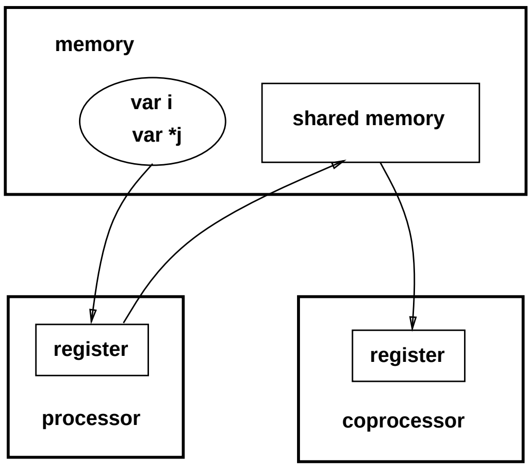

Figure 2: Transport of variables

Figure 3: Moving a BSB to hardware

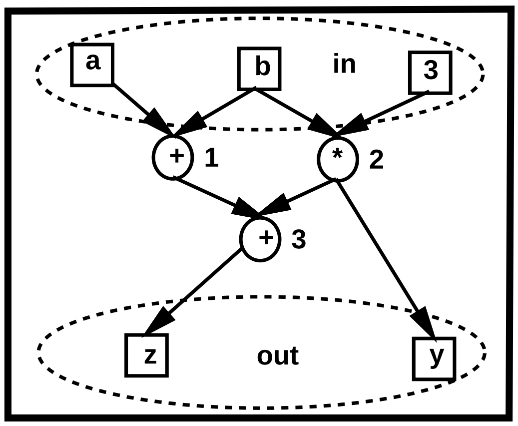

Figure 4: A simple BSB with one DFG

References

[BeHeEr93] Th. Benner,J. Henkel and R. Ernst,
Internal Representation of Embedded Hardware–/Software–Systems,
Innsbruck, Codes/CASHE '93, May 1993.

[Cyp89] Cypress Semiconductor,
The Cypress Semicondcutor RISC 7c600,
Cypress data book, 1989.

[ErHe92] R. Ernst and J. Henkel,
Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction,
Estes Park, Colorado, Handout from Int'l Workshop on Hardware-Software Co-Design, Oct. 1992

[ErHeBe93] R. Ernst, J. Henkel and Th. Benner,
Hardware/Software Co-Synthesis for Microcontrollers,
IEEE Design & Test of Computers (accepted), Dec. 1993.

[GuMi92] R.K. Gupta and G.D. Micheli,
System-level Synthesis using Re-programmable Components,
EDAC'92 IEEE Computer Society Press, pp. 2–7, 1992.

[Ha92] C. Hardewig,
Schnelle Ein-/Ausgabe für RISC-Proessoren,
Institut für Datenverarbeitungsanlagen, TU Braunschweig, PhD thesis, 1992.

[HBYSEG93] J. Henkel, Th. Benner, W. Ye, N. Serafimov, R. Ernst and G. Glawe,
COSYMA: A software–oriented Approach to Hardware/Software Co-Design,
The Journal of Computer and Software Engineering (accepted), 1993.

[HoEr93] U. Holtmann and R. Ernst,
Speculative computation for coprocessor synthesis,
Cambridge, Massachusetts, ICCD 1993, Oct. 1993.

[MiKuMa90] G.D. Micheli, D.C. Ku and F. Mailhot,
The OLYMPUS Synthesis System for Digital Design,
Design & Test Magazine, 37–53, Oct. 1990.

[MoYo87] N. Mort and S.S. Young,
Identification and Digital Control of a Turbocharged Marine Diesel Engine,
IFAC 87, 10th. World Congress an Automatic Control, Vol.3, pp. 250–253, 1987.

[PaSh91] C. Y. Park and A. C. Shaw,
Experiments with a Program Timing Tool Based on Source Level Timing Schema,
IEEE Trans. on Computer, pp. 48–57, May 1991.

[Po90] R. Potasman,
Percolation based synthesis,
Proc. of 27th. ACM/IEEE Design Automation Conference, pp. 444–449, 1990.

[PuKo89] R. Puschner and Ch. Koza,
Calculating the Maximum Execution Time of Real-Time Programs,
The Journal of Real-Time Systems, Kluwer Academic Press, pp. 159–176, 1989.

[WoWoDu92] N. Woo, W. Wolf and A. Dunlop,
Compilation of a single specification into hardware,
Estes Park, Colorado, Handouts from Int'l Workschop on Hardware-Software Co-Design, Oct. 1992.

[YErBeHe93] W. Ye, R. Ernst, Th. Benner and J. Henkel,
Fast Timing Analysis for Hardware-Software Co-Syntheis,
Cambridge, Massachusetts, ICCD 1993, Oct. 1993.