Adaptation of Partitioning and High-Level Synthesis in Hardware/Software Co–Synthesis

Jörg Henkel, Rolf Ernst, Ullrich Holtmann, Thomas Benner
Institut für Datenverarbeitungsanlagen
Technische Universität Braunschweig
Hans–Sommer–Str. 66
D–38106 Braunschweig
Henkel@ida.ing.tu–bs.de
GERMANY

Abstract

Previously, we had presented the system COSYMA for hardware/software co-synthesis of small embedded controllers [ErHeBe93]. Target system of COSYMA is a core processor with application specific co–processors. The system speedup for standard programs compared to a single 33MHz RISC processor solution with fast, single cycle access RAM was typically less than 2 due to restrictions in high-level co–processor synthesis, and incorrectly estimated back end tool performance, such as hardware synthesis, compiler optimization and communication optimization. Meanwhile, a high-level synthesis tool for high-performance co–processors in co-synthesis has been developed. This paper explains the requirements and the main features of the high-level synthesis system and its integration into COSYMA. The results show a speedup of 10 in most cases. Compared to the speedup, the co–processor size is very small.

1 Introduction – the COSYMA system

This paper presents a significant improvement of the hardware/software co-synthesis system COSYMA by using a specialized high-level synthesis system. COSYMA has been developed for the design of small embedded controllers (**COSY**nthesis of e**M**bedded **A**rchitectures), and first results have been published in [ErHeBe93].

While this section gives an overview, section 2 describes the features of the high–level synthesis system BSS. Section 3 summarizes the results gained by automated partitioning with COSYMA and synthesis by BSS. Finally section 4 gives a conclusion.

COSYMA is an experimental system for the co–design

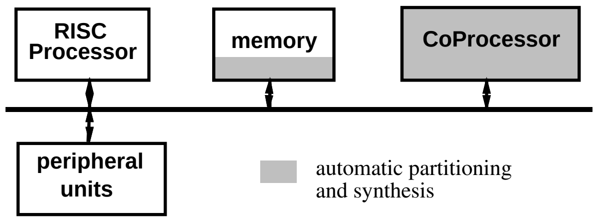

Figure 1: Target architecture of COSYMA

of small embedded real–time systems. The target architecture (figure 1) consists of a standard RISC processor core (we use the SPARC architecture with 33MHz clock and floating point co–processor as implemented by LSI Logic), fast RAM with single clock cycle access time and an automatically generated application specific co–processor. Processor and co–processor communicate through shared memory using a CSP type protocol (communicating sequential processes). The COSYMA design flow is shown in figure 2. The input is a real–time system description in C^x, a superset of the C language, which is enhanced by time constraints, communication, processes and some user directives to the co–synthesis process. This input description is translated to an extended syntax graph (ESG). The syntax graph is extended by a data flow graph for each basic block and the global control flow. An automated partitioning process is one of the key problems in hardware/software co–synthesis as opposed to co–design. COSYMA can partition *functions* or *basic blocks* including basic blocks with function calls. The hardware/software-partitioning in COSYMA is solved with simulated annealing ([OG89]). The approach is software–oriented, i.e. the simulated annealing starts with an all software solution and

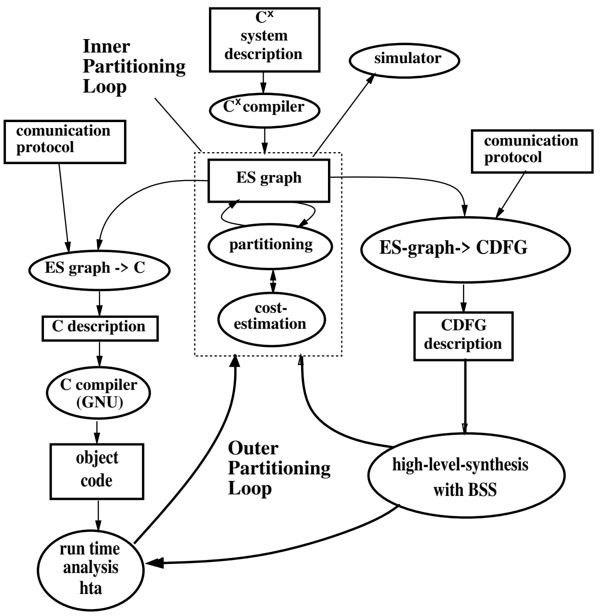

Figure 2: The design flow of the COSYMA experimental system

"extracts" hardware iteratively [ErHeBe93] until all time constraints can be met. This process is called the inner partitioning (loop). There is just one other comparable co–synthesis approach [GuCoMi92]which starts from the opposite point with mainly an all hardware solution and then uses a constructive rather than a stochastic algorithm. Because the partitioning cost criteria – circuit performance and area – are not known before synthesis and compilation have been executed, cost estimation is used instead. This estimation considers the potential speedup and communication costs. Circuit overhead is implicitly encoded by the optimization goal to implement as few as possible system functions (basic blocks, functions) in hardware.
The incremental speedup of a basic block (function) b extracted to hardware is *estimated* as:

$$\Delta c(b) = w \cdot (t_{HW}(b) - t_{SW}(b) + t_{com}(Z) - t_{com}(Z \cup b)) \cdot It(b) \tag{1}$$

where $\Delta c(b)$ is the estimated decrease in execution time, w is the weight factor to control simulated annealing [ErHeBe93], $t_{HW}(b)$ is the estimated co–processor execution time of b, $t_{SW}(b)$ is the estimated execution time of b on the processor and $t_{com}(Z)$ is the estimated processor–coprocesor communication time, given the current set Z of basic blocks on the co–processor. $It(b)$ is the number of iterations on b. All estimations are desired previous to

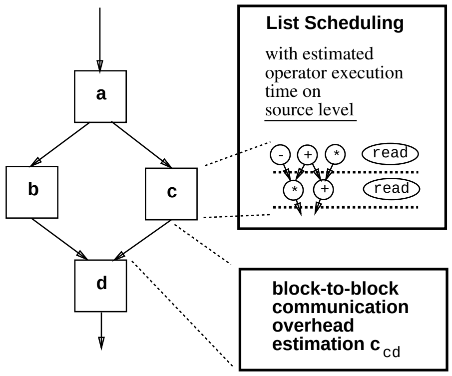

Figure 3: Estimation for partitioning

partitioning because simulated annealing needs fast cost function computation.

- $t_{SW}(b)$ is *estimated* with a local source code timing estimation. Estimation inaccuracy results from data dependent instruction execution times (SPARC: e.g. *mult*, *div*), optimization and register allocation in the compiler. Alternatively, trace data from processor simulation could be used (not in this paper).

- $t_{HW}(b)$ is *estimated* with a local list scheduling (figure 3) on b using the execution time (number of clock cycles) for each operator in high–level synthesis. The number of function units to be used is restricted by the user.

- $t_{com}(Z \cup b)$ is *estimated* by data flow analysis of *adjacent* basic blocks in the control flow. A global data flow analysis for each Z and b is too computation intensive. For shared memory, communication costs are proportional to the number of variables to be communicated: Let $C_{a,b}$ the number of variables flowing from a to b for the case that a and b are adjacent blocks, and 0 otherwise. At fixed transfer costs t_{trans} the communication costs $t_{com}(Z \cup b)$ are:

$$t_{com}(Z \cup b) = t_{com}(Z) - (\sum_{a \in Z} C_{a,b} - \sum_{d \in \cup \neg Z} C_{d,b}) \cdot t_{trans} \tag{2}$$

  Here $\neg Z$ is the set of blocks which are *not* on the co–processor.

- The weight w is choosen such that it drives the estimated system execution time T_S towards the required execution time T_c with a minimum number of basic blocks in the co–processor:

$$w = sign(T_C - T_S) e^{\frac{T_C - T_S}{T}} \tag{3}$$

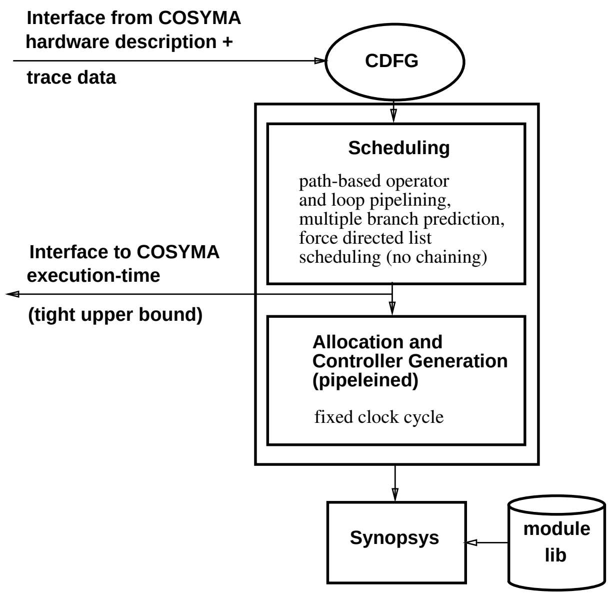

Figure 4: The design flow of the BSS HL-synthesis system

Those parts which are selected to be implemented in software are translated to a C program (using the GNU compiler) thereby inserting statements for communication with the co–processor. To minimize overhead, communication is now based on a global data flow analysis rather than a local analysis as used for estimation. The rest is translated to the input language of the high–level synthesis system, again inserting communication statements, and an application specific hardware is automatically generated.
Last step in the design flow is a fast timing analysis of the whole system [YErBeHe93]. If the time constraints are not fulfilled the design process is executed once more with changed parameters. We call this the outer partitioning loop. It is not automated yet, but requires user interaction. The outer loop is not considered in this paper. For more details of COSYMA see [ErHeBe93].

2 A Synthesis System for Co–Processor Generation

So far COSYMA has reached system speedups of typically less than 2 due to restrictions in the high-level synthesis system and incorrectly estimated back-end tool optimization, such as hardware synthesis, compiler optimization and communication optimization. Also, the synthesis times were in the order of several hours of CPU time on a SPARC 10/41. Iteration over synthesis in the outer loop was hardly practical under these circumstances. After the timing analysis had been cut to a few seconds [YErBeHe93], synthesis time is the co–synthesis bottleneck.

As a first step towards higher performance, a specialized co–processor synthesis system was developed. These were the requirements:

- *High speedup*
  It should exploit parallelism over loop boundaries, because most of the system parts moved to hardware are loops or contain functions with loops. It should accept hardware restrictions.

- *Short synthesis turnaround time in the outer partitioning loop*
  Short turnaround can not be achieved when RT–level or logic synthesis is included in the loop. Still, timing accuracy is required, otherwise the co–processor might need a longer clock cycle after implementation which would destroy the overall performance and force the processor to a slower speed, as well, synchronization. This could be achieved, if the clock timing is predictable during scheduling. To avoid synchronization the clock rate should be adapted to the clock rate of embedded RISC processors, which is 30-50 MHz. As we know from logic synthesis, controller timing can not be guaranteed at this speed. So, besides data path pipelining for complex operations, controller pipelining is useful. Operation chaining is too difficult to estimate, so it is omitted.

The co–processor synthesis system BSS (Braunschweig Synthesis System) is developed to fulfill these requirements. It performs operator pipelining, loop pipelining and speculative computation with multiple branch prediction ([HoEr93a]). The Scheduling is path–based with force directed scheduling along (predicted) paths [Ho93]. Because this is very fast, scheduling results are available after a few seconds of CPU time.
An upper bound for the run time is provided, because in exceptional cases, where several loops are executed in parallel, scheduling changes dynamically.
Input description of BSS is a CDFG. Trace data is required for timing analysis and speculative computation. A simple allocation follows scheduling (see figure 4). Currently, BSS only supports operations on 32 bit operands. Controller architecture was a major issue, and the solution is presented in ([HoEr93a]). The controller is optimized with the Synopsys logic synthesis system.

In our experiments, BSS always created co–processors with the desired clock rate of 33 MHz (the clock rate of the SPARC) for a $1.0\mu m$ standard cell technology (ES2). It should be pointed out that the high clock rate seems to be possible because the co–processor implements a small part of the C code only and because of controller pipelining.

Benchm.	loC	BSBs	constr.	moves	time (sec)
smooth	95	5	2.0	436700	41
		7	5.0	343500	35
		9	10.0	500000	52
trick	240	5	10.0	523900	74
key	1421	11	2.0	625500	304

Table 1: Partitioning procedure

Benchm.	loC	BSBs	geq	constr.	speedup
smooth	95	5	18320	2.0	4.39
		7	20260	5.0	6.50
		9	22160	10.0	9.65
trick	240	5	24070	10.0	9.59
key	1421	11	11900	2.0	2.66

Table 2: Automatically gained partitioning results

BSS has been integrated in COSYMA. In the following, we will give results for the use of BSS showing that the speedup has reached a level, where it is economically interesting.

3 Results

The experiments investigated the *system speedup*[1] and performance of BSS, the properties and suitability of simulated annealing and cost function for partitioning, and the relevance and accuracy of cost estimation. We selected typical benchmarks for small embedded systems

The *smooth*–benchmark implements a filter that smoothes the edges of a digital image. The benchmark *key* is part of an HDTV studio equipment and computes the parameters for an HDTV chromakey mixer. *Trick* is a small part of a program for professional online trick animation, which has been implemented in a rack–size system. These benchmarks have been selected because of their hard real–time constraints and their manageable size of about 100 to 1500 lines of C code (*loC* in tables).

Table 1 shows the size of the benchmark (loC), the number of basic blocks (BSB: basic scheduling block) moved to hardware, the user defined time constraint (*constr.*), the number of moves performed by simulated annealing and the computation time on a SPARC 10/41. First, the *smooth*–benchmark has been partitioned for 3 design points: required speedups of 2.0, 5.0 and 10.0. Intuitively, the number of moves of the simulated annealing for a 2.0 speedup should be less than for a 5.0 speedup but this is not the case (436700 moves for a 2.0 speedup and only 343500 for a 5.0 speedup). An investigation revealed that the annealing could not find a number of BSB, such that T_S is close to T_C (see equation 3), and because the number of blocks is small, each move changes costs significantly because of the step at $T_S = T_C$ and cooling was slow. This effect is the larger the more the individual blocks contribute to the system speed, i.e. the smaller the benchmark is.
This can also be seen from the results of the partitioning process of the benchmarks *trick* and *key*: it should be expected that the number of moves increases with the complexity of the benchmark (95 lines of C code for the *smooth*–benchmark and 240 and 1421 lines for *trick* and *key*). That is not the case because, here, it is much more easier for the annealing algorithm to find a set of BSB's such that T_S is close to T_C and then cool down. In all cases, only a few BSBs, mostly loops operating on arrays, have been moved to the co–processor. The speedup shows that simulated annealing has selected a few time critical parts. As a consequence, the co–processor controller is small, which is fovorable to BSS' hierarchical controller structure, which is basically a magic extension of [KuMi91]. The total execution times for the partitioning algorithm is at most 5 minutes.
The estimations are rather close to the actual results. Table 2 shows that for half of the benchmarks the deviation is less than 25 % and no actual circuit is significantly slower than estimated [ErHeBe93]. Before BSS, we typically found that the actual circuit was much slower than estimated. The table shows the size of the co–processor in gate equivalents for the ES2 $1.0\mu m$ standard cell library, without controller size, which is 10 - 20 % of the overall size and depends on Synopsys performance (for controller size see [HoEr93a]). The last column shows the actual speedup. The smooth example shows the effect of "reusing" the co–processor function units for different blocks: Even though the speedup rises by a factor of more than 2, the co–processors size increases only by about 20%.

As seen in table 2, we can reach speedups up to 10 – including the communication overhead between hardware and software. The co–processor size is in all cases less than 25k gate equivalents (geq), not including the controller.

The estimated speedup is almost always less than the actual speedup and never significantly larger. There are two reasons. First, list scheduling only considers basic blocks and, therefore, is inferior to the path based (force directed) list scheduling approach in BSS, which considers potential parallelism on paths through adjacent blocks. Second, communication estimation only considers adjacent blocks. In the key benchmark, a speedup of 2.0 was estimated (table 2) and 2.66 was obtained. Looking at the code, we found that loops were separated.

[1]The system speedup is defined as the relation between the execution time of an all–software–solution and the execution time of the complete hardware/software solution including also communication time.

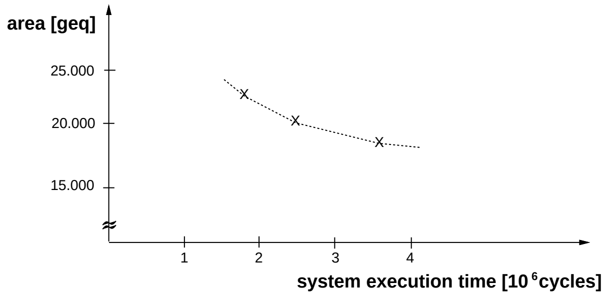

Figure 5: Design space of the *smooth*–benchmark

So, communication estimation which only analyzes adjacent blocks, did not recognize that there is a reduction in communication overhead if both loops are moved to hardware as one segment and the block stayed in software.

Figure 5 shows the area/runtime tradeoff for the *smooth*–benchmark. Here, the user performed a what-if analysis by changing the desired system speedup.

The turnaround times through the whole system are very small: for almost all benchmarks less than one hour could be reached.

Looking at the schedule, we found that, for higher speedups, the co–processor performance was limited by the memory bandwidth (2 cycle read/write i.e. 67MByte/sec). So, chaining does not seem to be a major issue for the benchmark cases, unless memory bandwidth can be increased. We conclude that memory design is the key topic for further speedups.

4 Conclusion

To improve hardware/software co-synthesis, a co–processor synthesis system, BSS, has been developed that is tailored to the requirements of the co-synthesis process. The results show a much higher speedup than achieved with a general purpose synthesis system while BSS consumes much less computation time in an iterative partitioning process. Realistic benchmarks helped to interpret the relation of estimation and actual results. In particular, the execution time of a BSS–generated co–processor is known very early in the synthesis process and turned out to be smaller than the estimations, such that the hardware/software system fulfills the time constraints. BSS speedup is typically constrained by the 67MByte/s memory bandwidth of the system. Having studied the "inner" partitioning process which is based on estimations, we currently work on the "outer" partitioning loop where the partitioning shall be automatically adapted to the actual speedup. The problem here is the convergence of the overall partitioning process and the cost parameter adaptation.

References

[ErHeBe93] R. Ernst, J. Henkel, Th. Benner, *Hardware-Software Cosynthesis for Microcontrollers,* IEEE Design & Test of Computers, pp. 64–75, Dec. 1993.

[GuCoMi92] R.K. Gupta, C.N. Coelho, G.D. Micheli, *Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components,* Proc. of DAC'92, pp. 225–230, 1992.

[GuMi92] R.K. Gupta, G.D. Micheli, *System-level Synthesis using Re-programmable Components,* Proc. of EDAC'92, pp. 2–7, 1992.

[Ho93] U. Holtmann, *High–Level Synthese mit BSS für den Einsatz im Hardware/Software Codesign (High–Level Synthesis with BSS for Usage in Hardware/Software Codesign),* Internal Report 931126-1, Institut f. Entwurf Integrierter Schaltungen, Technical University of Braunschweig, Germany, 1993.

[HoEr93a] U. Holtmann, R. Ernst, *Experiments with Low-Level Speculative Computation Based on Multiple Branch Prediction,* IEEE Trans on VLSI, Vol. 1, No. 3, Sep. 1993.

[KuMi91] D.C. Ku, D. De Micheli, *Constrained resource sharing and conflict resolution in Hebe,* Elsevier, INTEGRATION, the VLSI journal 12, pp. 131–165, 1991.

[OG89] R. Otten, P. van Ginneken, *The Annealing Algorithm,* Kluwer, 1989.

[YErBeHe93] W. Ye, R. Ernst, Th. Benner, J. Henkel, *Fast Timing Analysis for Hardware-Software Co-Synthesis,* Proc. of ICCD 1993, IEEE Society Press, pp. 452–457, 1993.