Fast Timing Analysis for Hardware-Software Co-Synthesis

W.Ye, R.Ernst, Th.Benner, J.Henkel
Technische Universität Braunschweig

Institut für Datenverarbeitungsanlagen, Hans-Sommer-Str. 66
38106 Braunschweig, Germany, wei@ida.ing.tu-bs.de

Abstract

At the current time, an iterative approach seems to be best suited for hardware/software partitioning in hardware/software co-synthesis with time constraints. To check the timing constraints the iteration loop contains a timing analysis. Only computation time intensive RT-level simulation provides sufficient timing precision for complex processor architectures. We present a hardware/software timing analysis, which comes close to the precision of an RT-level simulation in a fraction of the computation time and, thus, removes a bottleneck from iterative hardware/software co-synthesis. We present some results for our co-synthesis system COSYMA.

Keywords: Run Time Analysis, Embedded Systems, Hardware-Software Co-Design, Hardware-Software Co-Synthesis.

1 Introduction

Recently, hardware-software co-design has gained some attention as a promising way to design process improvement and to a reduced system design time. Hardware-software co-synthesis tries to automize the design process by combining synthesis and compiler technology with techniques to partition hard- or software functions. While some approaches are targeted to a very quick system implementation [ChRa92, WHLG92, BuVe92], in particular for prototyping, there are also co-synthesis approaches [GuMi92, ErHe92, WWD92, BaRo92] aiming at cost optimization under timing constraints. We will restrict the term hardware-software co-synthesis to such optimizing systems because they follow the same optimization objectives as hardware synthesis.

In the next chapter, we will explain why iterative partitioning seems to be best suited to hardware-software co-synthesis. We will expalin why *speeding up the timing analysis of the target system is key to practical iterative partitioning under timing constraints*. Chapter 3 scrutinizes the problems of fast timing analysis, and chapters 4 and 5 introduce our hybrid timing analysis approach. Chapter 6 and 7 give results and conclusions.

2 Co-Synthesis with iterative hardware-software partitioning

Given a target hardware system consisting of processors and application specific hardwired functions, a particular problem of co-synthesis is hardware-software partitioning. The partitioning must anticipate the compiler and synthesis results to be able to partition under cost optimization and time constraints.

Systematic and precise estimation of high-level synthesis results is an unsolved problem, especially with transformations such as tree height reduction and percolation based synthesis [Po90] altering the data flow. The synthesis tool might not even be fixed but there could be different synthesis tools used for different parts of a system. Even a precise estimation of software timing is hard in the context of optimizing compilers and complex RISC architectures, as we will show later.

Furthermore, hardware-software communication overhead must be considered. Our experiments with coprocessor generation have shown that it is reasonable to partition with fine granularity, at the basic block or statement level (simple examples from manual co-design: floating point coprocessors, vector coprocessors). In this case, communication overhead can be significant.

So, at the current time, there is little hope for sufficiently precise estimations, and *iteration over the software implementation and synthesis* - at least the high-level transformations and scheduling - *seems to be the only choice for cost optimization in cosynthesis*, in particular if hard timing constraints are involved.

Fig. 1 outlines our system COSYMA [ErHe92] for the co-synthesis of small embedded architectures, such as microcontrollers consisting of a processor and application specific hardware. The embedded system is described in C^x, a superset of C with parallel processes and timing constraints. The description is translated to an extended syntax graph (ES graph). A simulator is provided for verification and system profiling. Those parts of the system going to software are translated to C and are extended by statements for communication with hardware components before they are compiled to object code. At the moment, we only use a SPARC RISC processor core [Cy89]. Those parts going to hardware are translated to HardwareC, the language of the synthesis system OLYMPUS from Stanford [MiKuMa90].

Hardware-software partitioning is executed in a nested loop. In an inner loop, partitioning is executed using simulated annealing based on *cost and timing estimation.* The partitioning starts with an all-software solution moving parts of the system to hardware functions until all time constraints are satisfied.

The estimations are corrected with an outer loop including the allocation and scheduling steps of high-level synthesis and a run time analysis of the resulting hardware-software system. The run time analysis is provided with the allocation and scheduling data (and clock cycle) of the hardware as well as with the object code.

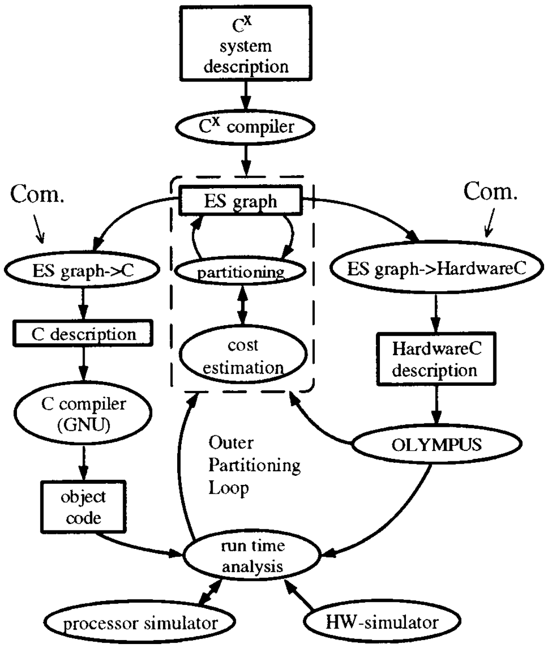

Fig. 1: COSYMA system

We developed our own processor simulator for run time analysis and target system verification, which accurately models program execution on the Cypress SPARC processor [Cy89] (and some others) at the *register-transfer (RT) level* (modelling all internal operations and states) with a single clock cycle as minimum time unit. A second simulator, Mercury, is available for simulation of the synthesized hardware. For realistic examples [ErHe92], *run time analysis with simulation* takes in the order of minutes to *several hours for processor simulation* (see results) and *several days for Mercury hardware simulation,* both on a dedicated SPARC 10/41 with 64 MBytes of main memory. This is unacceptable for iterative partitioning. As a comparison, high level synthesis scheduling&allocation and simulated annealing in the inner loop need typically less than an hour.

So, the run time analysis dominates the computation time of the partitioning system. This is not a peculiarity of COSYMA, but is considered generic for iterative partitioning with timing simulation.

The COSYMA system is now operable for coprocessor synthesis in a preliminary version.

3 The run time analysis problem

As seen, iterative co-synthesis requires much faster timing analysis. At the level of granularity at which we are partitioning (function, basic block, statement), an accurate modelling of processor instruction execution and hardware behavior is necessary to precisely account for the hardware-software communication overhead. This overhead is significant, even for simple communication mechanisms [ErHeBe93]. For synchronous communication RT level architecture modelling and simulation are *sufficient* but they are also *necessary*:

- Instruction execution in RISC processors is a complex interplay of several functional units. In particular pipelining leads to program dependent instruction execution times through pipeline interlocks. Fig. 2 shows the throughput of the last pipeline stage of a SPARC processor with 5 pipeline stages for a diesel engine control process compiled with the GNU C-compiler. The throughput is measured at the last pipeline stage (the write-back stage) and is averaged over an interval of 5 clock cycles ("filter width"). The throughput shows wide variations and, even worse, coprocessor communication would require additional load and store instructions which lead to further pipeline interlocks. The SPARC architecture is still rather simple compared to newer superscalar architectures with scoreboarding, such as the new Motorola 88100 [DiAl92].
- Another problem of timing analysis are data dependent instruction execution times of functions, basic blocks and statements, on the level of assembly instructions and on higher levels. Data dependent instruction execution times are a problem of the smaller microprogrammed processors but also of RISC processors, when instructions are not implemented in a particular processor and are translated to function calls with data dependent execution times, such as integer division in our SPARC processor (20 to 200 clock cycles).

To summarize, the timing analysis must consider the processor architecture at the RT-level and data dependencies.

So, as a primary goal, we would like to *reach the accuracy of an RT-level simulation* for the software and hardware parts, but in a much shorter time. Only as a secondary goal, we would welcome any progress towards worst case timing verification.

During partitioning the *overall system function remains fixed*, only the distribution of software and hardware functions changes. This could be used for *preprocessing*.

We will now split the problem in two parts, software run time analysis and hardware-software run time analysis. First, we will show that known techniques are not appropriate to even solve the software analysis problem alone and present a hybrid approach to drastically speed up software run time analysis. Then, we will show that this hybrid approach can also solve the hardware-software analysis problem in the form of a very precise estimation.

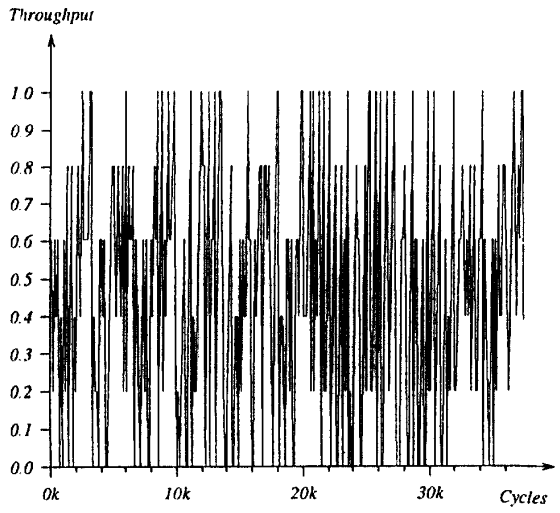

Fig. 2: Pipeline throughput (diesel engine ctrl.)

In this paper, we will focus on *timing requirements for process run times*. With simulation, the these timing conditions are checked for each input stimulus pattern at the end of the corresponding process execution. Usually, one cannot identify a single "worst case" input stimulus pattern because, when a time critical part is moved to hardware, the worst case might change. Furthermore, we assume that if the system uses caches (infrequent in small embedded real-time systems), cache effects are deterministic and covered by simulation.

4 Software timing analysis

There are many software timing estimation and analysis techniques in use today. *Program Profiling* [GrKeKu83, BaLa92] measures or samples the execution frequency of statements and the computation time spent in these statements, but only accumulated for program execution. *Program tracing* [La93] keeps track of the sequence of all executed statements or even their data references. Both require a simulator or the target architecture to be available for program execution. Because the target architecture is not available during iteration, (RT-level) simulation would remain in the loop.

Formal approaches can be used to cut software analysis times. Some of them use FSM modelling [CoRo83] or Petri-Nets [LeSt87]. Approaches to analyzing parallel tasks are usually limited to the level of tasks or program segments [LeSt87, CoRo83] based on timing estimations for tasks or segments and communication overhead. More precise approaches to single process analysis work on the level of individual program statements [PaSh91] or the assembly language [PuKo89, Mok89]. In [PaSh91], high level language statements are executed on the target processor and the derived timing is then used to estimate execution times. These approaches do not consider inter-instruction dependencies such as pipelining or data dependencies.

There is little hope that there could be a precise assembly or programming level timing estimation suitable to fine grain partitioning for pipelined and superscalar RSIC processors.

5 The hybrid timing analysis approach

The problem of simulation times as compared to formal analysis is repeated execution of the same statements in loops and function calls. To overcome the repeated execution of statements, we developed an approach, where a program is only simulated once and formal analysis is used in the sequel. We, therefore, call it a hybrid timing analysis (HTA) approach.

The first step is the identification and labelling of corresponding fork and join points in the program flow of the system input description in C^x using standard techniques. If we assume a structured input description with nested fork and join points, which we force the programmer to follow, each fork point has exactly one join point while join points may have several corresponding fork points (We removed the *go to* instruction from our C^x - compiler. Break operations need a particular treatment which is not explained here). The fork and join points mark the basic blocks of the input description. Accessing the compiler's debug information, the labels are transferred to the assembler level. The linker provides the label addresses for the object code, which are stored for the following simulation.

Now, RT-level simulation is executed controlled by a profiler/tracing function which exactly protocols the frequency of execution of a block A of the *object code*, f_A, (a basic block in the input description can consist of many basic blocks in the object code) and the overall time spent in each basic block, t_A. What is new is the separate protocol of the time spent in executing two adjacent blocks A and B with pipeline overlap. More precisely, t_{AB} is the time interval where instructions of A are still in the pipeline while B already started to execute. The time t_{AB} and the frequency of this transition f_{AB} are stored in a table. Simulation is data true, such that data dependent execution times and even data dependent block overlap are modelled correctly.

This system tracing and profiling based on simulation is done *once* as a *preprocessing step before iteration* on the initial all-software solution. No hardware simulation is necessary for preprocessing. Tracing and profiling is recorded for each process input stimulus pattern individually.

For the further steps, a *control flow graph* is constructed with the assembly level basic blocks as nodes - weighted with the profile and timing information - and with edges, which are weighted with the basic block overlap timing. Fig. 3 shows an example. The graph weights contain the aggregate tracing data, such that the total process run time (for the given input pattern) is *sum of all times in the graph.* Other than in the usual longest path analysis, this sum takes all data dependencies of the simulation into account.

During iterative partitioning in COSYMA, source level functions, basic blocks or statements are moved to hardware. The new process run time t_s is simply the sum of all execution times:

$$t_s = t_{SW} + t_{HW} + t_{COM} - t_{HW/SW}.$$

t_{HW} is the total run time of the synthesized hardware for the given stimulus. Again, we want to abstract from the individual hardware function execution, which can be data dependent and are only interested in the total run time of hardware computation steps. This total time can be derived from the hardware scheduling and the profiling information. The synthesis scheduling defines which operations are scheduled in each control step which is executed in one clock cycle. Because all operations in a control step c are executed in parallel, the total time spent in c is the maximum number of iterations It_{OP} of any operation scheduled in c. The iterations for the corresponding basic block in the source code are given in the control flow graph. The communication profile is fixed during partitioning. Then, the overall time spent in hardware is just the sum:

$$t_{HW} = \sum_{c} \mathrm{Max}\,(It_{OP})$$

For operator pipelining, the iterations must be divided by the number of stages, for mutual exclusive operations, the iterations must be added (not in OLYMPUS) OLYMPUS uses *relative scheduling* with *anchors*. Using anchors, independent loops can possibly be scheduled in parallel with no synchronization except at the end. When such loops are data dependent, the HTA approach is not applicable. We did not encounter such presumably rare cases.

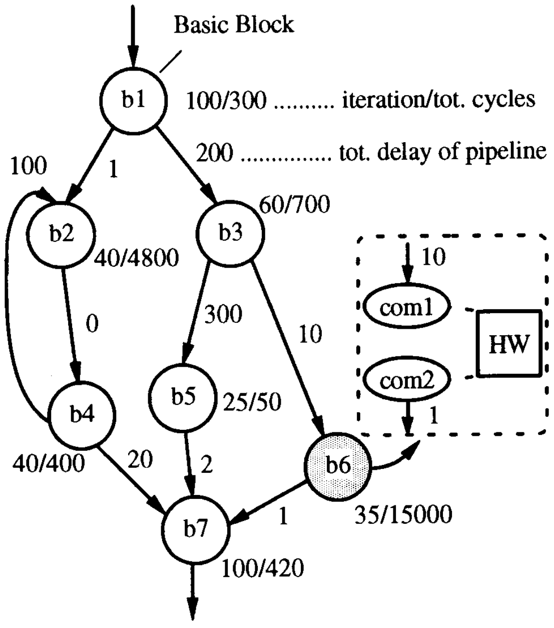

Fig. 3: Control flow graph for software timing analysis

t_{SW} is estimated from the control flow graph. When a source level basic block or function b has been moved to hardware, the corresponding join and fork points are missing on the assembly language level (debug information). Then the node is marked in the graph and the time t_b is subtracted from the previous time t_{SW}'. The timing of the data independent load/store instructions for communication t_{COM} is simulated and then added to the graph (fig. 3). So,

$$t_{SW} = t_{SW}' - t_b + t_{COM}$$

The overlap timing of b is still counted, which is a worst case assumption because t_{COM} also contains a pipeline load phase. When a basic block b has changed on the assembly code level, because either single statements have been moved to hardware or the compiler optimization has changed, it is locally resimulated, and the execution time t_b^* is multiplied by the number of block iterations It_b. Data dependencies are not correctly modelled in local simulation, but the possible error due to data dependencies in a single basic block is usually very small, except for data dependent execution times. We can also bound the error by a formal worst case analysis of the block execution giving time t_b^{WC}.

$$t_b = It_b * t_b^* \text{ ; Max. Err.} = It_b * (t_b^{WC} - t_b^*)$$

bench-mark	simulation		HTA (devt.=0)	worst case analysis	source code analysis
	simulated cycles	simulation time	simulation time	deviation	deviation
key	170545286	2688.4s	2.0s	961.8	
smooth	1781712	41.2s	0.3s	45.9	32.7%
trick	636450099	16007.0s	1.1s	782.3	
diesel	22403	1.4s	0.1s	673.1	1.6%
3d	1377	0.2s	<0.1s	100.1	121.0%

Tab. 1: Precision and computation time of analysis methods (without preprocessing)

$t_{HW/SW}$ is the overall time where hardware and software functions execute concurrently. In COSYMA, we currently use a simple mechanism with mutual exclusion, such that $t_{HW/SW} = 0$.

6 Results

First, we want to compare our results to formal analysis *without partitioning*. Tab.1 shows the execution times of benchmarks from several sources: HDTV studio equipment (key [Ri92], trick), a diesel engine control [MoYo87] (diesel), a and DSP-filters (3d, smooth). For each benchmark, we give three data. The first column shows the results of the RT-level simulation, the target processor clock cycles and the CPU time for simulation on a SPARC 10/41 and a single input pattern. The second column gives the CPU time for HTA after profiling and tracing. The results must be identical, because it uses the exact timing and profiling from the control flow graph. Then, a formal worst case analysis is executed without regarding data dependencies between basic blocks. The results are far off the real behavior, as checked by inspection in some examples. Obviously, worst case analysis is not usable for our purposes. To improve the precision of formal worst case analysis, the user is often asked to provide loop bounds manually [PaSh91,PuKo89]. The 4th column shows a source code timing analysis, only for the smaller examples, where the execution time of compiled high level statements was measured [PaSh91], and then these times where multiplied with the number of iterations taken from profiling. The large deviations are due to data dependent statement execution times and compiler optimizations. This supports our claim that source code level timing analysis is not sufficiently accurate.

Tab. 2 shows results for partitioned circuits. Currently, COSYMA can only partition the simpler of the examples automatically. The hardware timing analysis approach in HTA is exact. Therefore, we only compared the deviation in software and communication timing. In all cases, the deviation of HTA-estimation and simulation is less than 1% with analysis times of less than 1s per input stimulus pattern.

7 Conclusion

We presented an approach to fast and precise timing analysis in iterative hardware-software co-synthesis. It is a hybrid approach combining RT-level simulation and formal analysis. An initial simulation provides profiling data to account for data dependencies. The analysis is applicable even for complex processor architectures. The results show, that it reaches the timing precision of an RT-level simulation at a fraction of the computation time.

bench-mark	partitioned $T_{SW}+T_{COM}$				
	simulated		analysed		
	simulated cycles	simu-lation time	analysed cycles	ana-lysis time	devt.
diesel	14230	1.1s	15121	<0.1s	0.6%
smooth	1325409	40.2s	1436779	0.9s	0.8%
3d	1377	0.2s	1429	<0.1s	0.5%

Tab. 2: Results for partitioned circuits

Acknowledgement

This work has, in part, been supported by the Deutsche Forschungsgemeinschaft. Clemens Hardewig provided the fig. 2.

References

[GrKeKu83] S.L. Graham, P.B. Kessler, M.K. McKusick. An execution profiler for modular programs. Software-Practice and Experience, Vol.13, pp.671-685.

[BaLa92] Th. Ball, J.R. Larus. Optimally Profiling and Tracing Programs. ACM Sigplan Symp. Principles of Programming Lang., Albuquerque 92, pp. 59-70.

[La93] J. R. Larus. Efficient Program Tracing. IEEE Computer, May 93, pp. 52-61.

[Mok89] A. Mok et al. Evaluating Tight Execution Time Bounds of Programs by Annotations. Proc. IEEE WS Real-Time Operating Systems and Software, May 89, pp.74-80.

[ErHeBe93] R. Ernst, J. Henkel, Th. Benner. Hardware-Software Co-Synthesis for Microcontrollers. Accepted for IEEE Design&Test.

[BaRo92] E. Barros, W.Rosenstiel. A clustering approach to support hardware/software partitioning. Handouts from IFIP Workshop on Hardware-Software Codesign, Grassau, Germany, May 1992.

[BuVe92] K. Buchenrieder, C. Veith. CODES. A Practical Concurrent Design Environment. IEEE WS on Hardware-Software Co-Design, Estes Park, Colorado, Oct. 92.

[ChRa92] D.C.Chen, J.M.Rabaey. A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths, JSSC, Dec. 92, pp. 1895-1904.

[CoRo83] J. E. Coolahan, N. Roussopoulos. Timing Requirements for Time Driven Systems Using Augmented Petri Nets. IEEE Trans. on Softw. Eng.. Sep. 83, s. 603-616.

[Cy89] Cypress Semiconductor, Seminar Series 1989. The Cypress Semiconductor RISC 7C600.

[DiAl92] K. Diefendorff, M. Allen. Organization of the Motorola 88110 Superscalar RISC Microprocessor. IEEE Micro, April 1992.

[ErHe92] R.Ernst, J.Henkel. Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction. IEEE WS on Hardware-Software Co-Design, Estes Park, Colorado, Oct. 92.

[GuMi92] R.K.Gupta, G.D. Micheli. System-level Synthesis using Re-programmable Components. EDAC'92, Brussels, Feb. 92, pp. 2-7.

[LeSt87] N. G. Leveson, J. L. Stolzy, Safety Analysis Using Petri Nets. IEEE Trans. on Softw. Eng, May 87, pp. 386-397.

[MiKuMa90] G.D. Micheli, D.C. Ku, F. Mailhot et al.. The OLYMPUS Synthesis System for Digital Design, Design & Test Magazine, Oct. 90, pp. 37-53.

[MoYo87] N.Mort, S.S.Young. Identification and Digital Control of a Turgocharged Marine Diesel Engine, IFAC 87, 10th World Congress on Automatic Control, vol. 3, pp. 250-253.

[PaSh91] C. Y. Park, A. C. Shaw. Experiments with a Program Timing Tool Based on Source Level Timing Schema IEEE Trans. on Comp. May 91, pp. 48-57.

[Po90] R. Potasman et al., Percolation Based Synthesis, 27th DAC, 90, pp. 444-449.

[PuKo89] P. Puschner, Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. Real-Time Systems, Sep. 89, pp. 159-176.

[Ri92] Ch. Ricken. Optimierung der automatischen Einpegelung eines HDTV-Chromakey-Mischers. Master Thesis, Technische Universität Braunschweig, 92.

[WWD92] N.Woo, W.Wolf, A.Dunlop. Compilation of a Single Specification into Hardware and Software. IEEE WS on Hardware-Software Co-Design, Estes Park, Colorado, Oct. 92.

[WHLG92] P.Windirsch, H.-J. Herpel, A.Laudenbach, M.Glesner. Application-Specific Microelectronics for Mechatronic Systems. EURODAC 92, Hamburg, Sep. 92, pp.194-199.