Profiling in the ASP Codesign Environment

Matthew F. Parkinson and Sri Parameswaran

Department of Electrical and Computer Engineering
University of Queensland, St. Lucia 4072
Brisbane, Queensland, Australia
email : parkinso@elec.uq.edu.au

Abstract

Automation of the Hardware/Software Codesign methodology brings with it the need to develop sophisticated high-level profiling tools. This paper presents a profiling tool which uses execution profiling on standard C code to obtain accurate and consistent times at the level of individual compound code sections. This tool is used in the ASP Hardware/Software Codesign project. The results from this tool show that profiling must be performed on dedicated hardware which is as close as possible to the final implementation, as opposed to a workstation.

1 Introduction

Automated design methodologies in digital systems have until recently been limited entirely to the design of hardware. Automated Hardware/Software Codesign (HSC) offers a design methodology for a total system (ie. both hardware and software). The ASP (Automated Synthesis & Partitioning) codesign environment is an on going project at the University of Queensland on Hardware/Software Codesign.

For a totally hardware oriented design (eg. ASICs) the development time is prohibitive in bringing fresh and affordable products to the market. Equally restrictive is a totally software based solution which will perform slowly due to the use of a generalised computing architecture (ie. a RISC based microprocessor). This is where designing for a hybrid between a hardware and software based implementation can be of particular advantage.

A codesign methodology enables the specification of an algorithm totally in software. Through an automated design process the algorithm is optimally partitioned into both hardware and software, thus allowing the designer to be distanced from the hardware specific techniques of improving an algorithm's performance. This in turn allows the designer to concentrate on the algorithm's design.

Algorithm bottlenecks are usually limited to a small portion of the actual code. By converting these critical code segments into hardware, an ideal partitioning of the algorithm's execution into both hardware and software is achieved. An overview of the automated Hardware/Software Codesign methodology is briefly outlined in Figure 1. This automated partitioning process initially identifies the critical code segments within the software. These code segments are then used to provide a near optimal partition between hardware and software implementations. By next applying the process of synthesis to the partitioned code it is possible to achieve significant acceleration of the algorithm.

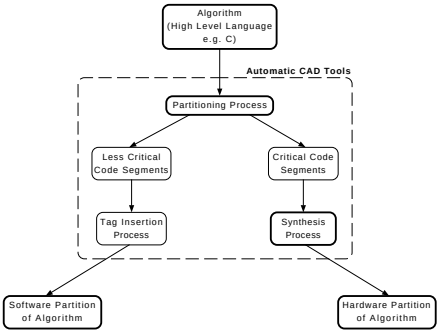

Figure 1 - Automated hardware/software codesign methodology

The automation of this partitioning process also permits the design to be independent of the final hardware required for execution. The actual hardware implementation is determined through cost and resource constraints. This easily allows the designer to take advantage of emerging technologies without the requisite redesign of the system from the ground up. An example of a proposed hybrid architecture is outlined in Figure 2.

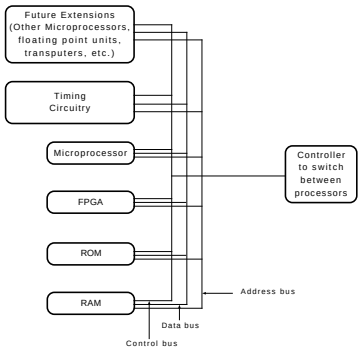

Figure 2 - Hybrid architecture configuration

Acceleration of the algorithm can be achieved by implementing critical sections on dedicated hardware accelerators or on FPGAs. Execution on FPGAs is performed by converting the high-level standard C language description of the hardware partition into VHDL [2] code. By then passing the VHDL code through the MEBS [3] package, we have both a hardware description of the partition, as well as feedback on the cost in terms of chip area, and execution time. This cost is related to implementing the algorithm on a XILINX FPGA.

To achieve an ideal partitioning between hardware and software execution of a given algorithm, a profile of the C code must be generated. Ideally this profile should outline execution tendency and data transfer within the various sections of the C code under consideration. Until now, most methods have tended toward estimation, or simulation techniques to achieve these profiles. Smith [20] presents an overview of execution graphs from which performance is estimated. Ein-Dor and Feldmesser [21] give a method for predicting performance on a system, while techniques are developed by Shaw [22, 23, 24] for reasoning about the execution time of processes. The co-synthesis system COSYMA [19] presents a processor simulator for run time analysis and target system verification at the register transfer level.

Problems with worst-case behaviour, excessive simulation time, as well as pipelining anomalies exist with these current techniques. The ASP profiler is able to construct profiles of C code in near real-time.

In order to achieve profiles of C code execution that were as close to reality as possible under the ASP system, a new profiling methodology was developed. As our system is specified in C, a tool was constructed to profile the C code description of an algorithm in order to determine the appropriate partitioning between hardware and software.

The profiling tools described here consist of both hardware and software components. Section 2 outlines the software developed, while Section 3 details the hardware aspects of the design. Section 4 compares results obtained using the profiling tools on several platforms.

For a complete description of the HSC process as it is applied above, please refer to [1], [4] and [5].

2 Profiling software

There are several key factors which affect the ability to perform both accurate and consistent profiling of any given C code. Those attributable to software are outlined below along with the methods used to attain results which are more than satisfactory for the partitioning process of Hardware/Software Codesign.

2.1 Timing resolution

One major factor in attaining satisfactory profiling results is the timing resolution available. On standard workstations a limit of one-hundredth of a second is imposed, even though the software times returned are specified in micro-seconds. The rounding is a direct result of the hardware limitation of the timers.

Not only is this resolution unacceptable, but there is also the problem of accumulating time from other processes, running on the same machine. This is overcome by calling only those timing routines which activate the timers while our process's context is active. Even so, on a standard workstation, the timers are not able to be stopped at any given point within the code, which is a requirement for obtaining consistent and meaningful times.

With this approach, the times returned for many sections of code are zero. Couple this with the recursive nature of loops, and an accumulative effect is seen. This often renders results which don't reflect the type of code being executed.

In the context of Hardware/Software Codesign, it is crucial to determine the true extent of execution of any code segment. With the embedded systems architecture of our design, we attain the needed resolution directly with the hardware that will finally execute it. The resolution we obtain with this method is exact, to the level of individual clock cycles, as the software is executed on our target architecture. Futher aspects of the hardware design are discussed in Section 3.

2.2 Software tagging

Implementing a timing resolution at the level of individual clock cycles doesn't by itself satisfy the requirements of the Hardware/Software Codesign process. While it does exhibit a timing resolution five orders of magnitude better than present methods, it can only be of benefit if the appropriate software techniques are employed. Discussed below are the software tagging techniques used to take full advantage of the hardware design.

Another resolution limitation identified in the profiling process is that of segmentation granularity. This is the minimum resolution imposed on the identification of candidate segments within the C code being profiled. The profiler limits segmentation granularity to compound statements. These are further referred to as compound code sections.

In order to identify the compound code sections within the original C code being profiled and further, to obtain detailed timing information about these sections, software tagging is used before the final compilation process. Software tagging consists of two stages, **parsing** and **insertion**. These two processes are now discussed further.

The first stage consists of parsing the original C code to identify procedures and functions and record variable usage. The location of compound statements is also recorded at this point. These include standard loop constructs, conditionals, and all statements enclosed in braces.

The locational information obtained from this first parsing stage is used in the insertion process of the second stage. The second stage consists of inserting profiling code in the original C source. This provides an execution trace and permits an iteration count to be maintained, with the identity of each section being recorded. Timing information is also recorded for each compound code section as each section is entered and again on leaving. It is also seen that a previous aim, that of being able to stop the timer, while these details are being recorded, is now able to be fulfilled.

Stopping of the timers during house-keeping is essential to maintaining consistent timing results. Time allocated to these tasks is not uniform from one invocation to the next. If included in overall timing figures, bias is introduced, with smaller routines potentially returning times containing large variation.

By itself the timing of house-keeping tasks appears to be of little consequence when applied to the HSC process, due to the small execution time under consideration. However, coupled with the recursive, and loop-based nature of algorithms, it leads to potentially meaningless results, regardless of the high resolution of the timers employed. It effectively short-circuits an overall aim of profiling, that of obtaining consistent results from one invocation to the next.

The insertion process avoids timing unnecessary tasks by placing code to inhibit the timers on entry to each new compound code section. After house-keeping is performed, the timers are re-enabled. All time attributed to these last two operations is deducted in the recording process of house-keeping.

The initial parsing stage identifies all points in the original C code where a given compound code section may be terminated. Consequently, the second stage also inserts code to stop the timers, wherever the execution trace may leave each compound code section. Once again, house-keeping is performed at this stage.

With the parsing and insertion stages completed, the resulting code is next compiled in preparation for execution on the target architecture. This final process yields a dynamic model which is representative of the execution flow structure, timing information and data flow of the algorithm during execution.

The aim of execution profiling is to build this dynamic model. For a statistically accurate model, the designer must ensure that any testing data is representative of the general data set the algorithm will encounter during normal operation. In this way, every branch within the algorithm may be exercised.

An example of a typical piece of code containing a single entry point and multiple exit points, along with inserted profiling calls, is detailed in Figure 3.

2.3 Comparison with other systems

Previous efforts [9] such as the 'proof of concept' system PRISM-I have failed to address the areas of the design process discussed above, preferring to rely on the hand-selection of code to be converted to hardware. The resolution of the system presented in [9] is also limited to converting entire functions into hardware, compared with the compound code sections handled automatically by the system presented in this paper.

Commercially available profilers were found to be inadequate for the ASP design process. They either had a poor timing resolution (e.g. GCC's profiling option), provided no structural information about the code, or were not portable over varying target architectures.

In designing a profiler the opportunity was available to develop an entirely theoretical approach through the use of simulation. The other alternative was that of performing real-time analysis of algorithms as they executed on various target architectures. The advantages of execution profiling over simulation arise from the nature of the ASP project.

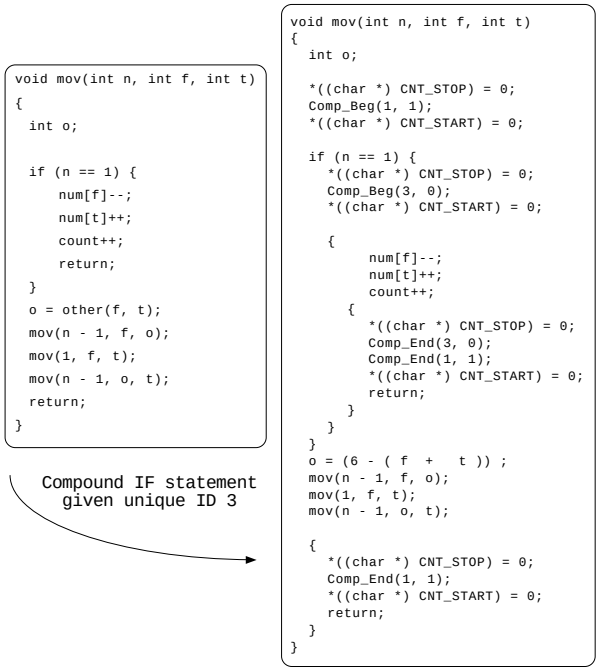

Figure 3 - Profile call insertion

During this project many target hybrid architectures will be encountered. The prohibitive expense associated with developing models for the simulation of these architectures was crucial. The profiler merely requires a cross compilation to change the architecture under consideration. Another inhibiting feature of simulation is its slow execution speed, which in turn results in a lengthy development cycle time.

With the accuracy, consistency, and speed of execution with which profiling fulfils the requirements of the ASP project, the alternative of simulation is no longer a consideration.

3 Profiling hardware architecture

There are several hardware features of the design which attribute to the success of execution profiling. These are outlined below.

3.1 Timing architecture

By designing dedicated timers into the architecture performing profiling, timing is suspended and resumed through the software being profiled. Advance knowledge of the time associated with toggling the timing status ensures exact times are recorded in all cases.

Clocking rate of the timers is identical to that of the processing element under consideration. For microprocessor execution, timer clocking rate is identical to CPU speed. This ensures an exact representation is maintained at all times during profiling.

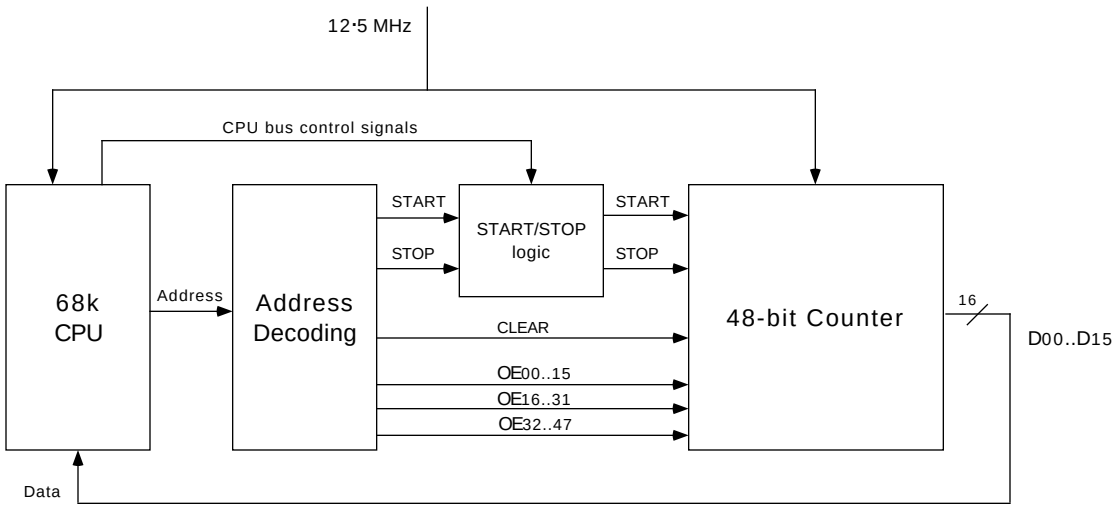

Figure 4 - Timer interface

It is possible that the code used to start and stop the timer's will be compiled into assembly instructions which vary throughout the code being profiled. This is a direct result of differences in register allocations. These variations are of the order of clock cycles, and appear infrequently. The extent to which profiling results are effected is extremely minimal. Consistent profiling results are still produced from one run to the next.

The timer interface is extremely simple. The software is required to clear, start, stop, and read the counter values of the timer. Some of these software operations are detailed in Figure 3. The timer interface is shown in Figure 4.

A 48-bit timer is presently used to avoid overflow problems. This ensures continuous profiling at our current CPU speed of 12.5 MHz for a period of over 8 months.

3.2 Timing accuracy and consistency

The critical timing sections occur when the clock is stopped and once again when it is started. By themselves these timing situations are not difficult. But, when coupled with external events, such as DRAM refreshes, and the subsequent bus requests associated with these events, they become a serious timing concern.

It is necessary to deduct all time associated with DRAM refresh cycles, in order to maintain consistent and accurate times for all code on each invocation of the profiler. And importantly, any overlap between the starting and stopping of the timers, and DRAM refresh bus requests, must be resolved.

These objectives have been achieved, resulting in a profiling system which successfully returns identical times for the same C code, on separate invocations of the profiler. Not only has this consistency of times been achieved, but an accuracy of timing has been achieved whereby all code is timed with absolute precision. The times returned by the profiler for any section of C code are accurate down to the level of each individual clock cycle.

The timing is both exact in terms of accuracy and perfectly consistent in terms of the variation from one invocation of the profiler to the next.

4 Results

The software techniques employed for profiling (the timing of code sections, as well as procedures) were also tested on the standard timing hardware of a workstation. The results from these software techniques for profiling were superior to the results from profiling tools which were available on the system. They do however lack the accuracy and consistency obtained with the dedicated hardware developed for the ASP project. Consequently, the usefulness of profiling on a workstation is limited to an initial measure of a program's execution tendency.

Ten common benchmark programs were tested with the profiling tools which have been developed. The benchmarks were tested both on the 68k based dedicated hardware designed for the ASP project, as well as on standard workstations. The workstations included a Sparcstation 10, and a 68k based workstation. The only difference between the profiling performed on the dedicated hardware, and that performed on the workstations is in the timers used to record the execution time at a given point in the code.

The workstation is unable to turn off it's timers. Consequently, the time for house-keeping tasks is included in its timing. Various random events, such as DRAM refresh, may also be included in these times. The timing routines used are those which only run in process virtual time (ITIMER_VIRTUAL - under "man getitimer"). The timer only runs when the process is executing. The frequency of the workstation timers was 100 Hz, compared with 12.5 MHz for the dedicated hardware.

The example benchmarks profiled on the workstation were each run 20 times, with a standard deviation being calculated across the 20 invocations. A standard deviation was calculated for each compound code section, of each example benchmark.

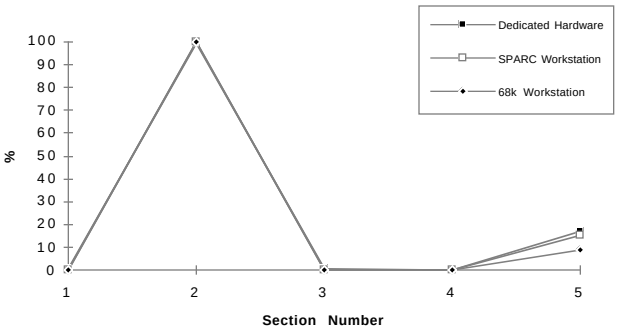

Figure 5 - Towers of Hanoi benchmark

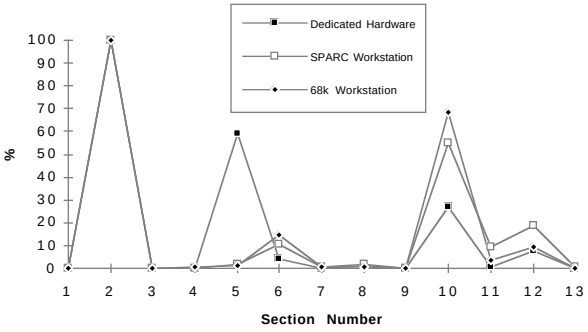

Figure 6 - Heapsort benchmark

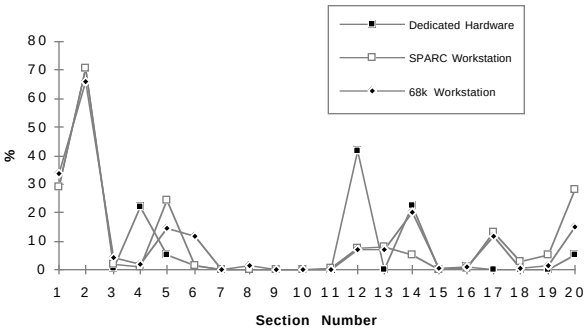

Figure 7 - Fast Fourier Transform benchmark

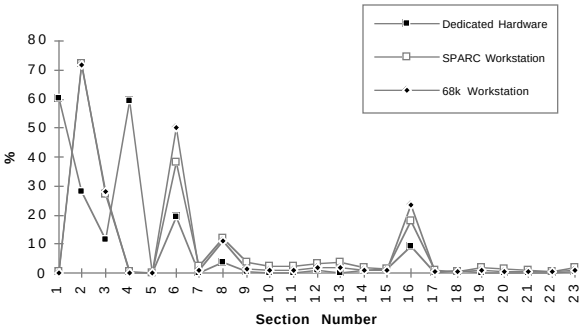

Figure 8 - ADPCM Coder/Decoder benchmark

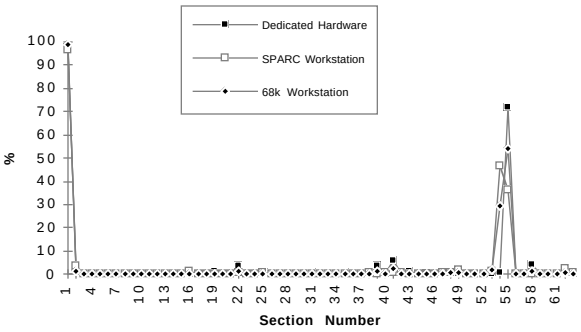

Figure 9 - Livermore Loops benchmark

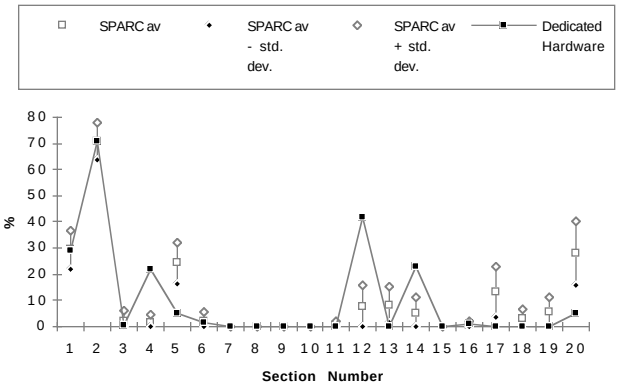

Figure 10 - Fast Fourier Transform - SPARC

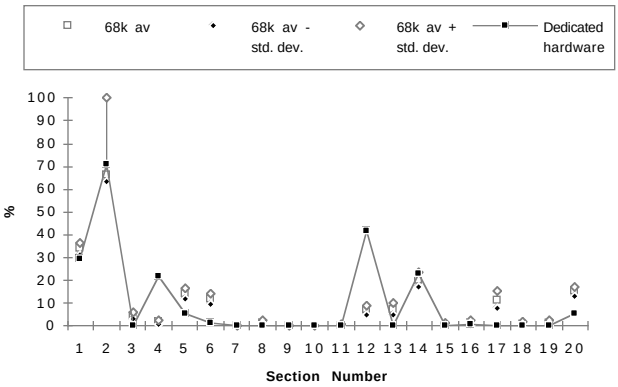

Figure 11 - Fast Fourier Transform - 68k

The charts of Figures 5 through 9 represent a comparison between the exact percentages, as calculated on the dedicated hardware, compared with execution on a SPARC workstation, and a 68k based workstation. Figures 10, 11 show the workstation results, represented by the average percentage calculated, along with marks one standard deviation either side of this average, for the FFT Benchmark. The average and standard deviation results are calculated over 20 independent runs on the workstations, and are calculated on the % of overall time taken. The first few percentages on each graph represent procedure execution time.

5 Future research

As our Hardware/Software Codesign project achieves its goals with respect to synthesizing hardware, the profiling techniques presented will be extended to timing sections of code as they execute in hardware. This will allow further decisions to be made regarding the suitability of placing particular sections of code in either the hardware or software partitions of a given design.

Timing granularity is another profiling issue associated with the timing of hardware execution. As it is possible for the synthesized hardware to execute with a faster clock than the software partition as it runs on a microprocessor, the granularity required for the hardware partition may have to be increased to attain the desired comparisons.

The possibility of variations in timing, through starting and stopping the timers, will require further attention.

6 Conclusions

We have presented a tool which profiles standard C code on several platforms. This tool was developed as part of the ASP Hardware/Software Codesign project at the University of Queensland. As such, it facilitates our need to separate a software specification into both hardware and software executable components. This is partially achieved through the timing of individual compound code sections with both accuracy and consistency.

Profiling was performed on 3 platforms. These consisted of our own dedicated hardware containing a 68k CPU, a SPARC processor based workstation and a 68k based workstation. It was shown that the profiling of benchmarks on workstations was not accurate, and indeed returned results with large standard deviations, when performed over 20 runs.

A further comparison between times obtained from the 68k based dedicated hardware and the 68k based workstation revealed significant differences. Based on the fact that timing performed on our dedicated hardware is both accurate and consistent, we therefore conclude that it is both practical and necessary to perform profiling on the same architecture that will be used in the final implementation of the Hardware/Software Codesign process.

7 Acknowledgement

The authors wish to acknowledge the work of Paul Taylor on some parts of the software.

8 References

[1] Matthew F. Parkinson, Paul M. Taylor, and Sri Parameswaran, "An Automated Hardware/Software Codesign (HSC) using VHDL," Proceedings of the First Asia Pacific Conference on Hardware Description Languages and their Applications (APCHDLSA '93), December 1993.

[2] P. Ashenden, "VHDL Cookbook," - Internet

[3] "MEBS," - Internet

[4] Matthew F. Parkinson, Paul M. Taylor, and Sri Parameswaran, "A Profiler for Automated Translation of Signal Processing Algorithms into High Speed Hardware/Software Hybrid Architectures," Proceedings of Microelectronics '93, October 1993.

[5] Matthew F. Parkinson, Paul M. Taylor, and Sri Parameswaran, "C to VHDL Converter in a Codesign Environment," Proceedings of VHDL International Users Forum, May 1994.

[6] GNU CC, Reference Manual - Internet

[7] YACC, Reference Manual - Internet

[8] B. Bose, M. E. Tuna, and S. D. Johnson, "System Factorisation in Codesign," Proceedings of the 1993 IEEE International Conference on Computer Design (ICCD '93), October 1993.

[9] P. M. Athanas, and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," Computer IEEE, pp. 11-18, March 1993.

[10] R. Ernst, J. Henkel, "Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction," Handout from First Int'l Workshop on Hardware-Software Codesign, Estes Park, Colorado, 1992.

[11] R. Gupta, CC. Coelho, and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," Proc. DAC, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 225-230.

[12] D. C. Chen, J. M. Rabaey, "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths," JSSC, pp. 1895-1904, December 1992.

[13] P. Windirsch, H.-J. Herpel, A. Laudenbach, M. Glesner, "Application-Specific Microelectronics for Mechatronic Systems," EURODAC 92, Hamburg, pp. 194-199, September 1992.

[14] K. Buchenrieder, C. Veith, "CODES. A Practical Concurrent Design Environment," IEEE Workshop on Hardware-Software Codesign, Estes Park, Colorado, October 1992.

[15] R. K. Gupta, G. D. Micheli, "System-level Synthesis using Re-programmable Components," EDAC'92, Brussels, February 1992, pp. 2-7.

[16] N. Woo, W. Wolf, A. Dunlop, "Compilation of a Single Specification into Hardware and Software," IEEE Workshop on Hardware-Software Codesign, Estes Park, Colorado, October 1992.

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

[18] W.H. Wolf, "Hardware-Software Co-Design of Embedded Systems (and prolog)," Proceedings of the IEEE, July 1994, Vol. 82, No. 7, pp. 965-989.

[19] W. Ye, R. Ernst, Th. Benner, J. Henkel, "Fast Timing Analysis for Hardware-Software Co-Synthesis," Proc. ICCD'93, IEEE Computer Society Press, 1993.

[20] C. U. Smith, "Performance Engineering of Software Systems," Reading, MA, Addison-Wesley, 1990.

[21] P. Ein-Dor and J. Feldmesser, "Attributes of the Performance of Central Processing Units : A Relative Performance Prediction Model," Commun. ACM, Vol. 30, No. 4, pp. 308-317, April 1987.

[22] A. C. Shaw, "Reasoning about Time in Higher-Level Language Software," IEEE Trans. Software Engineering, Vol. 15, No. 7, July 1989.

[23] A. C. Shaw, "Deterministic Timing Schema for Parallel Programs," Proc. 5th Int. Parallel Processing Symp., IEEE Computer Society Press, 1991, pp. 56-63.

[24] C. H. Park and A. C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," IEEE Transactions on Computers, pp. 48-57, May 1991.