Cycle-approximate Retargetable Performance Estimation at the Transaction Level

Yonghyun Hwang Samar Abdi Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine, 92617-2625
e-mail: {yonghyuh, sabdi, gajski}@uci.edu

Abstract

This paper presents a novel cycle-approximate performance estimation technique for automatically generated transaction level models (TLMs) for heterogeneous multi-core designs. The inputs are application C processes and their mapping to processing units in the platform. The processing unit model consists of pipelined datapath, memory hierarchy and branch delay model. Using the processing unit model, the basic blocks in the C processes are analyzed and annotated with estimated delays. This is followed by a code generation phase where delay-annotated C code is generated and linked with a SystemC wrapper consisting of inter-process communication channels. The generated TLM is compiled and executed natively on the host machine. Our key contribution is that the estimation technique is close to cycle-accurate, it can be applied to any multi-core platform and it produces high-speed native compiled TLMs. For experiments, timed TLMs for industrial scale designs such as MP3 decoder were automatically generated for 4 heterogeneous multi-processor platforms with up to 5 PEs under 1 minute. Each TLM simulated under 1 second, compared to 3-4 hrs of instruction set simulation (ISS) and 15-18 hrs of RTL simulation. Comparison to on-board measurement showed only 8% error on average in estimated number of cycles.

1. Introduction

Heterogeneous multiprocessor platforms are increasingly being used in system design to deal with growing complexity and performance demands of modern applications. However, choosing the optimal platform for a given application and the optimal mapping of the application to the platform is crucial. Such system level decisions require early and accurate estimation of performance for a given design choice. Cycle accurate models do provide accuracy but may not be available for the whole platform. Furthermore, cycle accurate instruction set simulation models (ISS) for processors and RTL models for custom HW are too slow for efficient design space exploration. Although ISS models use an instruction set abstraction, the mapped application is interpreted by ISS at run time, which slows down simulation. Our proposed native compiled timed TLMs bypass the problem of interpreted models using a preprocessing step that annotates the application code basic blocks with accurate delay estimates. Hence, our TLMs provide performance estimates that are cycle approximate but simulate at speeds close to reference C code.

The performance annotation of application code can be done at various levels of accuracy by considering different features of a processing element (PE) such as operation scheduling policy, cache size and policy and so on. The PE model is a set of these parameter values. While generating the timed TLM, each basic block in the application is analyzed to compute the estimated number of cycles needed to execute it on the given PE. The number and combination of parameters used to model the PE, determine the accuracy of the estimation. Therefore, several timed TLMs are possible depending on the detail of PE modeling. The more detailed the PE model, the longer is the delay computation time. A tradeoff is needed to determine the optimal abstraction of PE modeling. In this paper, we consider operation scheduling policy, datapath structure, memory delay and branch delay as the most important parameters for PE modeling. We use an abstract bus channel based communication model [16] to manage the problem size.

The rest of the paper is organized as follows. In Section 2, we present a comparison of our technique with state of the art dynamic estimation techniques. In Section 3 we position TLM estimation framework as part of a system design methodology. In Section 4, we describe our estimation algorithms and software architecture. Experimental results scalability and accuracy with several multi-processor design examples are presented in Section 5. Finally, we wind up

the paper with conclusions and future work.

2. Related Work

There have been several efforts in early performance estimation of multiprocessor systems for the past 15 years. The approaches can broadly be categorized as static, semi-static and dynamic. Static approaches use analytical models of the platform architecture to compute delays for applications mapped to them. Semi-static approaches use source level profiling to gather application characteristics and use the application to platform mapping to generate performance estimates. Fully dynamic approaches, such as the one espoused in this paper, use platform models to generate timed executable model of the design that produces estimation data at run time. ISS and virtual platforms are popular examples of dynamic approaches.

Each estimation approach can be evaluated on the basis of speed, accuracy, abstraction level and generality. Speed and accuracy are natural concerns. Abstraction level is important because during early estimation, detailed models of the PEs may not be available. Finally, generality is relevant because a heterogeneous platform may have custom PEs. A purely SW estimation technique relies on instruction set abstraction and may not be applicable to such a platform.

In [12], a static estimation based on a designer-specified evaluation scenario is proposed. However, the estimation is not cycle level and is not applicable to custom HW. SW performance estimation techniques [9], [8], [2], [3], and [4] claim to provide estimation at transaction level, but they do not take into account the processor datapath structure. Unlike above techniques, [10] and [5] can take into account the datapath structure by using ISS, but the generated models are extremely slow. [14], [7], [13] provide fast system simulation models, but they are not retargetable and cannot consider custom hardware which makes then unscalable. Simplescalar [1] is a well known retargetable ISS that can provide cycle accurate estimation result but is several orders slower than TLM.

3. TLM Estimation Framework

Figure 1 shows TLM estimation framework in the context of a system design methodology, enabled by the Embedded System Environment (ESE) [6] tool set from the Center for Embedded Computer Systems (CECS), UC Irvine. Our TLM estimation tool is part of the ESE front-end. Designers may have several models with different abstraction levels to describe a multiprocessor system. TLMs offer fast simulation at a higher level of abstraction. On the other hand, Pin-Cycle Accurate Models (PCAMs) have detailed design implementation but simulate much slower.

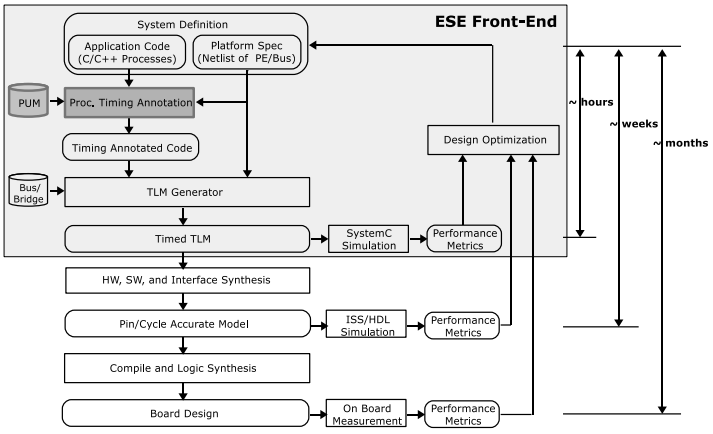

Figure 1. System Design Methodology

Designer generates TLMs from the design decisions to verify the system early in the design cycle. Then, TLMs go through cycle accurate synthesis step resulting in PCAMs. Using PCAMs, designers optimize and re-verify their design to meet given design constraints. After PCAMs are finalized, HW synthesis and SW compilation are performed to generate prototype board design. The time spent in TLM generation and test is in order of hours. In case of PCAMs and prototype board design, it is in order of weeks and months respectively. Therefore, design and verification with PCAMs is too slow to efficiently explore the design space. Our proposed design methodology applies performance annotation before TLM generation. The timed TLMs allow the designer to perform fast and early evaluation of design choices. This can shorten the system design cycle drastically, because design iteration with TLM simulation is in the order of few hours.

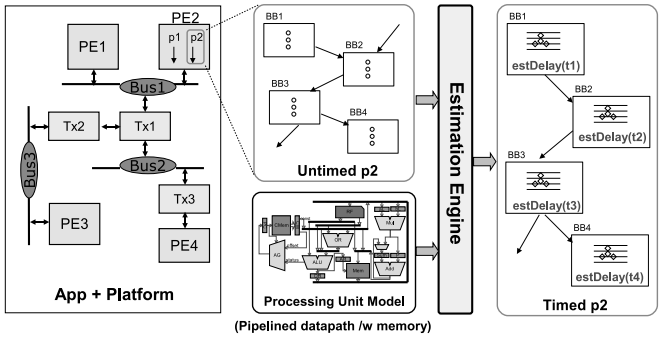

Figure 2. TLM timing Annotation

Figure 2 shows an overview of our timing annotation technique. The timing annotator takes design decisions at transaction level and analyzes processes in PE. Then, the application process is translated into control data flow graph(CDFG). This CDFG is fed into the estimation engine

along with the processing unit model (PUM). PUM characterizes the structure of PE and has the scheduling policy for the PE. The estimated timing delay for each basic block is added at the end of the basic block and the timing annotated process code is input to the TLM generator.

4. Estimation Tool Architecture

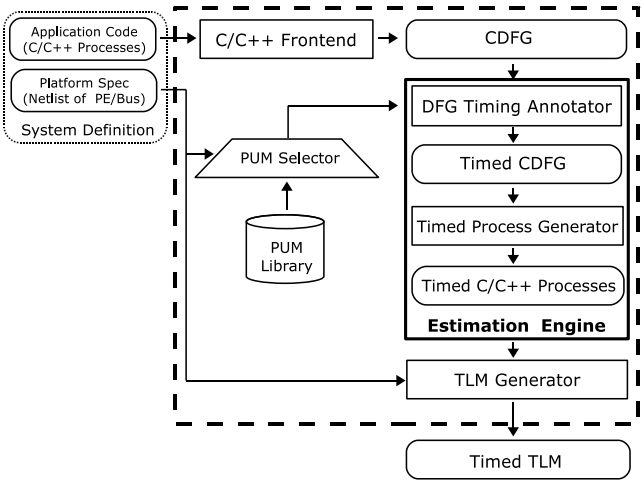

Figure 3. Estimation Tool Architecture

In this section, we discuss the architecture and implementation of our performance estimation tool. A C/C++ front-end is implemented using the LLVM compiler infrastructure [11] to parse application processes into CDFGs. For each basic block inside the CDFG, the corresponding DFG is fed to an estimation engine. The DFG delay is computed based on the PUM for the respective PE. The estimated delay is annotated into the CDFG data structure using LLVM source transformation API. Timed C code for the process is generated using the LLVM code generation API. Finally, the annotated C code is compiled and linked with a SystemC programming model of the platform to generate the simulatable timed TLM.

4.1. Processing Unit Model (PUM)

The PUM consists of the following data models:

1. **Execution model** consists of *scheduling policy* and *operation mapping table*. The *scheduling policy* define the operation scheduling algorithm used by the PE such as ASAP, ALAP, List etc. The *operation mapping table* keeps two flags: *demand_operand* and *commit_result* that specify the pipeline stages where the operation needs operand and commits the result, respectively. Furthermore, a usage table is associated with each operation pointing to the datapath unit and mode used by operation in each pipeline stage.

2. **Datapath model** is a set of *functional unit*s, and *pipeline*s. *Functional unit* has an id, type, quantity, possible operation modes, and delays for each operation mode. For example, ALU may have addition and multiplication modes with different delays. *Pipeline* model defines the function units used per stage. Multiple pipelines are allowed for superscalar architectures.

3. **Branch delay model** is a statistical model that store the branch prediction policy, cycles lost for misprediction and the average misprediction ratio.

4. **Memory model** is also a statistical model that stores the average i-cache and d-cache hit-rates and cache memory access latencies for a set of cache sizes. The external memory latency is also specified here.

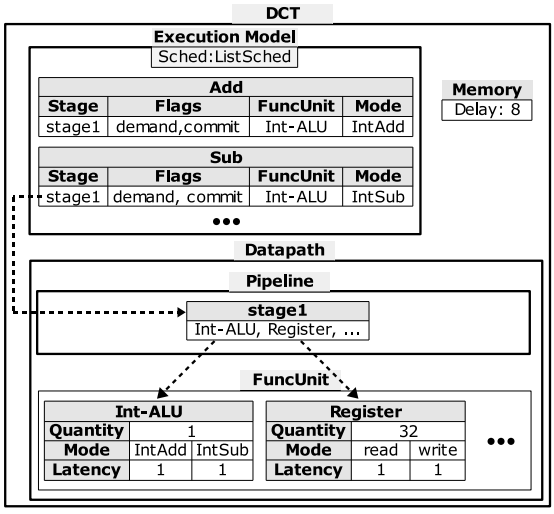

Figure 4. PUM example: DCT HW

Figure 4 shows the PUM examples for custom HW (DCT). DCT has a non-pipelined datapath and no memory hierarchy. The register files and block rams used for storage have a single cycle delay. The absence of a pipeline in DCT is modeled as an equivalent single issue pipeline with only one stage in its PUM. In Figure 5, the PUM of a MIPs-like microprocessor(MicroBlaze) is described. This PUM has configurable instruction/data cache and single issue pipeline. As shown in the examples above, PUM is flexible and general enough to describe not only configurable embedded processors but also custom IPs.

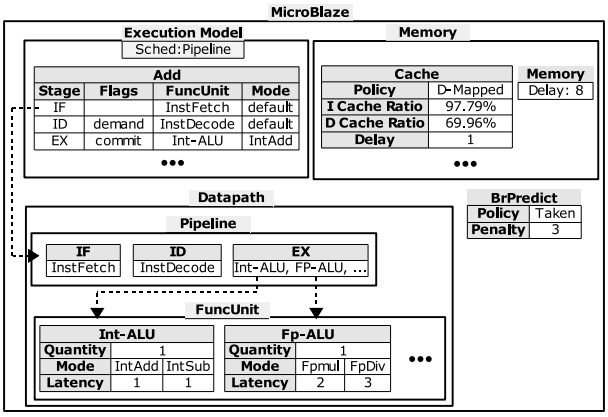

Figure 5. PUM example: MicroBlaze

4.2. Estimation Algorithm

The *DFG Timing Annotator* in Figure 3 computes the estimated delay for each basic block using the PUM. The estimated basic block delay has three components:

1. number of cycles for operation schedule,
2. delays due to cache-misses, and
3. branch misprediction penalties

In order to compute the scheduling delay, we simulate the DFG of the basic block on the execution model in the PUM while assuming optimistic cache behavior of 100% hit rate and no branch misprediction.

Algorithm 1 Optimistic Scheduling

```
1:  delay = 0
2:  c_set = { } // operations in pipeline
3:  d_set = { } // operations done
4:  r_set = {x| all operations in BB } // remaining operations
5:  initialize pipeline // list data structure for pipeline
6:  while |d_set| != # of BB operations do
7:     for all pipeline do
8:        d_set = d_set U advClock(pipeline)
9:        c_set = c_set - d_set
10:    end for
11:    for all pipeline do
12:       c_set = c_set U AssignOps(pipeline.stg1, r_set)
13:       r_set = r_set - c_set
14:    end for
15:    delay = delay + 1
16: end while
17: return delay
```

Algorithm 1 computes the scheduling delay for a basic block with optimistic assumptions. The basic idea of the algorithm is to simulate the scheduling behavior of the PE to compute the cycle delay for a single DFG. The PE behavior is simulated by function *advClock* (line 8) until all operations in the DFG are completed i.e. done set has all the operations on the basic block(line 6). The simulation is guaranteed to terminate because there are no cycles in the DFG.We start by initializing the pipeline data structure. In the first iteration, *advClock* does nothing because the pipeline is still empty. Function *AssignOps* assigns operations from the remaining set to the first stage of the pipeline, based on the *Operation Scheduling Policy* in PUM. The assigned operations are added to the current set (line 12).

Function *advClock* simulates the each pipeline stage as follows. A commit set of operations in the DFG is maintained. These are operations that have moved beyond their commit stage of the pipeline (marked by *commit* flag). Thus the results of these operations are ready. For all operations in a given stage, a counter keeps the remaining cycles for the operation in the current stage. For every call to *advClock*, all counters are decremented by one. If a counter reaches 0, it is advances to the next stage if the next stage in not a *demand* stage. For a demand stage, the data dependencies of the operation are checked. If all the dependencies are in the commit set (i.e. all operands are available and no data hazard), then the operation is advanced to the next stage. Finally, *advClock* returns the set of operations that are in their last stage and have remaining cycle counter value of 0. These operations are added to the done set (line 8). Finally, the scheduling delay is returned (line 17).

Algorithm 2 Compute BB Delay

```
1:  BB_delay = OptimisticSchedule()
2:  if PE is pipelined then
3:     BB_delay += BP_miss_rate * Br_penalty
4:  end if
5:  if PE has i-cache then
6:     BB_delay += # of BB Ops * (i_cache_miss_rate
       * i_cache_miss_penalty + i_cache_hit_rate *
       i_cache_delay)
7:  end if
8:  if PE has d-cache then
9:     BB_delay += # of BB Operands *
       (d_cache_miss_rate * d_cache_miss_penalty
       + d_cache_hit_rate * d_cache_delay)
10: end if
11: return ⌈BB_delay⌉
```

To incorporate the delays from cache miss and branch misprediction, Algorithm 2 is used. It uses the optimistic scheduling delay from Algorithm 1 and adds to it the product of branch misprediction rate and penalty values from

PUM (lines 1-4). A similar method is used to estimate i-cache and d-cache delays for the basic block. The summation of all the delays is rounded off and returned (line 11).

4.3. Timed TLM Generation

Once the delays for each basic block in each application process are estimated, we annotate them to the process source code. For this purpose, the estimation engine uses the LLVM compiler infrastructure [11] for addition of a *wait()* function call to each basic block. The annotation is performed on the internal CDFG data structure using the LLVM API. Recall the the original CDFG has been created by the LLVM parser. Finally, C code for the processes with the annotated wait calls is generated by LLVM code generator. This C code is then compiled and linked with a SystemC wrapper to generate the timed TLM executable.

The SystemC wrapper is simply a transaction level programming model of the platform. It consists of instantiated modules for each PE with interfaces to channels for respective buses. The bus channel provides abstract interprocess communication functions [16]. Each application process mapped to a PE is instantiated as a *SC_PROCESS*. The functions for the process are implemented in C code that was annotated by the estimation engine. The SystemC wrapper also carries the implementation of the *wait* function that is called at the end of each basic block. The process ID is passed as parameter to the *wait* call. The *wait* function keeps the accumulated delay for each process at any given time during the TLM simulation. At each interprocess transaction boundary, the accumulated delays are applied to the SystemC simulation using the *sc_wait()* function. We do not apply *sc_wait* after each basic block execution because it is an expensive function that forces the SystemC simulation kernel to reschedule simulation events. Inter-process transactions are the minimum granularity for applying *sc_wait* and this granularity is user controllable in the estimation engine.

5. Experimental Results

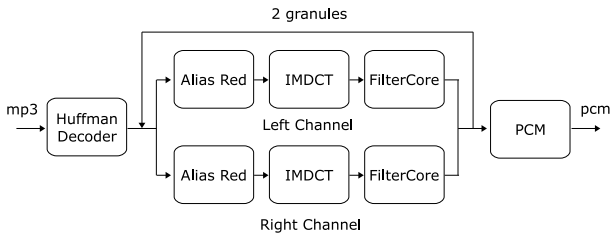

Figure 6. MP3 Decoder Application

To evaluate our estimation engine, we generated TLMs for 4 designs of MP3 decoder application as shown in Figure 6. The most computationally intensive functions are FilterCore and IMDCT. The first design was a pure software implementation on MicroBlaze [15] referred to as SW. In the second design, referred to as SW+1, the left channel FilterCore function was moved to a custom HW component. In the third design, SW+2, both FilterCore and IMDCT for the left channel were moved to custom HW components. Finally, in design SW+4, FilterCore and IMDCT from both channels were moved to custom HW components. Also, several instruction and data cache sizes for MicroBlaze were tried. The design goal was to reduce the decoding time for each MP3 frame without incurring significant area penalty.

Design	Anno.	TLM$_{func}$	TLM$_{timed}$	PCAM
SW	31.181 s	0.002 s	**0.004 s**	**15.93 h**
SW+1	49.841 s	0.006 s	**0.216 s**	**17.56 h**
SW+2	47.364 s	0.010 s	**0.253 s**	**17.71 h**
SW+4	71.108 s	0.012 s	**0.355 s**	**18.06 h**

Table 1. Scalability Results: Annotation and Simulation Time for Timed TLM

I cache /D cache	Board Cycles	ISS		TLM	
		Cycles	Error	Cycles	Error
0k/0k	27.22M	16.47M	39.48%	25.51M	6.27%
2k/2k	8.91M	7.28M	18.38%	8.32M	6.68 %
8k/4k	5.83M	5.62M	3.55%	5.55M	4.74%
16k/16k	4.41M	5.13M	-16.32%	5.02M	-13.83%
32k/16k	4.38M	5.11M	-16.60%	4.99M	-13.89%
Average	N/A		**18.86**%	N/A	**9.08**%

Table 2. Accuracy Results (SW only)

Timed TLMs were generated for each of the above designs using our estimation engine. ISS models were generated for the pure SW application only because fast cycle accurate C models were unavailable for custom HW components. PCAM models were developed by manually coding RTL for the custom HW components. Finally, the PCAMs were synthesized and downloaded to Xilinx FF896 board using ISE and EDK tools for on-board measurements.

Table 1 shows simulation time for the generated timed TLMs in comparison to purely functional TLMs, and PCAMs. ISS could only be used for the SW design and took 3.2 hours to complete simulation of 1 frame decoding. We also show the annotation time for the different designs. It can be seen that annotation time increases as HW components are added because custom HW units use a more complex operation scheduling policy than MicroBlaze. However, even for a complex design like SW+4, the annotation time is close to a minute. The simulation time for the timed TLMs is under a second like functional TLMs. In contrast, ISS and PCAM simulations are in the order of hours.

I/D Cache Size	SW+1			SW+2			SW+4		
	Board	**TLM**	**Error**	**Board**	**TLM**	**Error**	**Board**	**TLM**	**Error**
0k/0k	26235952	23874437	9.00%	24692474	20204483	18.18%	24673298	20082499	18.61%
2k/2k	7314936	7838744	-7.16%	5847507	6770896	-15.79%	6083583	6652535	-9.35%
8k/4k	5790118	5261430	9.13%	26235952	23874437	9.00%	4486701	4494908	-0.18%
16k/16k	4997837	4765105	4.66%	4310239	4196674	2.63%	4231982	4077381	3.65%
32k/16k	4375281	4737844	-8.29%	4239167	4172609	1.57%	4149428	4054285	2.29%
Average	**N/A**		**7.65%**	**N/A**		**7.97%**	**N/A**		**6.82%**

Table 3. Accuracy Results: Error % against Board Measurement

Table 2 show accuracy results for estimation with ISS and TLMs relative to actual board measurements. The cycle counts are in millions. It is interesting to note that average error in timed TLM estimation was actually half of ISS estimation error. This is because the MicroBlaze ISS available to us did not model memory access accurately enough.

Table 3 shows results for accuracy of the estimation designs with HW units compared to on-board measurement, using a timer. On an average the estimates were within 6-9% of board measurements, which is a very high degree of accuracy. Error rates did fluctuate for different cache sizes and we used absolute error values to compute averages. We could not get any conclusive results on the sensitivity of estimation to the statistical memory and branch prediction models in PUM. This is the focus of our future research.

6. Conclusions

We presented a technique and tool for cycle approximate retargetable performance estimation using TLMs. The tool is part of the Embedded System Environment (ESE) toolset that also includes SystemC TLM generation from graphical platform and application capture. Results with design of MP3 decoder application showed that our estimation engine is scalable to complex heterogeneous platforms and its estimation results are within 9% of actual board measurements. As a result ESE allows designers to experiment with different platforms and applications since timed TLMs are generated automatically for any design change. For future work, we plan to improve our PE data models by adding RTOS parameters. We also want to study the sensitivity of estimation results to our statistical memory delay and branch penalty models.

Acknowledgments
We would like to thank Pramod Chandraiah for reference code of MP3 decoder and Lochi Yu for the TLM generator. We would like to thank previous generations of CAD lab members at UC Irvine for contributions to basic principles that our work builds on.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[2] J. R. Bammi, W. Kruijtzer, and L. Lavagno. Software Performance Estimatioin Strategies in a System-Level Design Tool. In *CODES*, San Diego, USA, 2000.

[3] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-Level Execution Time Estimation of C Programs. In *CODES*, Copenhagen, Denmark, 2001.

[4] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration . In *DATE*, San Diego, USA, June 2004.

[5] M.-K. Chung, S. Na, and C.-M. Kyung. System-Level Performance Analysis of Embedded System using Behavioral C/C++ model. In *VLSI-TSA-DAT*, Hsinchu, Taiwan, April 2005.

[6] ESE: Embedded Systems Environment. "http://www.cecs.uci.edu/ ese".

[7] FastVeri (SystemC-based High-Speed Simulator) Product. "http://www.interdesigntech.co.jp/english/fastveri/".

[8] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance Estimation Framework for Early System-Level-Design using Fine-grained Instrumentation. In *DATE*, Munich, Germany, March 2006.

[9] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation. In *CODES*, Rome, Italy, May 1999.

[10] J.-Y. Lee and I.-C. Park. Time Compiled-code Simulation of Embedded Software for Performance Analysis of SOC design. In *DAC*, New Orleans, USA, June 2002.

[11] LLVM(Low Level Virtual Machine) Compiler Infrastructure Project. "http://www.llvm.org".

[12] J. T. Russell and M. F. Jacome. Architecture-level Performance Evaluation of Component-based Embedded Systems. In *DAC*, Anaheim, USA, June 2003.

[13] VaST: Virtual System Prototype Technologies. "http://www.vastsystems.com/solutions-architecture-systems.html".

[14] Xilinx. *Embedded System Tools Reference Manual*. 2005.

[15] Xilinx. *MicroBlaze Processor Reference Manual*. 2007.

[16] L. Yu, S. Abdi, and D. Gajski. Transaction level platform modeling in systemc for multi-processor designs. Technical Report CECS-TR-07-01, January 2007.