# A SW performance estimation framework for early System-Level-Design using fine-grained instrumentation

Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz,
Gerd Ascheid, Rainer Leupers, Heinrich Meyr

*Institute for Integrated Signal Processing Systems,*
*RWTH Aachen University, Germany*
*kempf@iss.rwth-aachen.de*

## ABSTRACT

*The increasing demands of high-performance in embedded applications under shortening time-to-market has prompted system architects in recent time to opt for Multi-Processor Systems-on-Chip (MP-SoCs) employing several programmable devices. The programmable cores provide a high amount of flexibility and reusability, and can be optimized to the requirements of the application to deliver high-performance as well. Since application software forms the basis of such designs, the need to tune the underlying SoC architecture for extracting maximum performance from the software code has become imperative.*

*In this paper, we propose a framework that enables software development, verification and evaluation from the very beginning of MP-SoC design cycle. Unlike traditional SoC design flows where software design starts only after the initial SoC architecture is ready, our framework allows a co-development of the hardware and the software components in a tightly coupled loop where the hardware can be refined by considering the requirements of the software in a stepwise manner. The key element of this framework is the integration of a fine-grained software instrumentation tool into a System-Level-Design (SLD) environment to obtain accurate software performance and memory access statistics. The accuracy of such statistics is comparable to that obtained through Instruction Set Simulation (ISS), while the execution speed of the instrumented software is almost an order of magnitude faster than ISS. Such a combined design approach assists system architects to optimize both the hardware and the software through fast exploration cycles, and can result in far shorter design cycles and high productivity. We demonstrate the generality and the efficiency of our methodology with two case studies selected from two most prominent and computationally intensive embedded application domains.*

## 1. INTRODUCTION

The rapidly growing and highly competitive embedded system market is putting extreme demands on system architects to deliver high performance, low power solutions within very short design and development cycles. Such conflicting demands of short time-to-market and high performance can only be met by incorporating *reusability*, and hence, *flexibility* in the overall design. Since sufficient flexibility and reusability can only be provided through programmable processor cores, designers in recent times have started employing an increasing number of Multi-Processor System-on-Chip (MP-SoC) platforms combining several programmable devices. Therefore, the amount of software and its impact on the overall system performance is on the rise, and is expected to remain so for some time. The increasing importance of software has led to a situation where software bottlenecks must be taken into account while designing the underlying MP-SoC communication architecture and selecting the programmable cores.

Traditional system design tools and methodologies are extremely inadequate to address these challenges. These hardware-software co-design and co-exploration with (potentially) several ISS (Instruction Set Simulators) and HDL (Hardware Description Language) model of communication architectures are too slow to execute and too difficult to change. Fast simulation of the communication networks is possible by task graph based simulators, but such modeling frameworks do not permit validation of task functionalities. Moreover, intra-task memory accesses are completely kept out of consideration in such cases.

The recent and future MP-SoC platforms will require intensive synergy and interaction between software and hardware development from the beginning of design cycles. Designers will like to start with a piece of software, coarsely partitioned into several tasks, and a very basic communication architecture, and then refine it iteratively depending on simulation results. In each step they will like to *split, merge and change* different tasks, *reassign* them to different processing elements, *adapt* the communication architecture accordingly, *validate* the overall system functionality and see the effects. The *primary requirement* for such a design-flow is efficient software simulation within *System-Level-Design (SLD)* environments with accurate task latency and memory access modeling. Unfortunately most SLD tools are not suitable for such design-flow, since they use task models that are either very coarse and simplistic (e.g statistical task graph simulation tools) or extremely slow and complicated (i.e. ISS).

This paper presents a novel MP-SoC design framework that fills this void by incorporating fast software performance estimation techniques in a *Transaction Level Modeling (TLM)* environment. The *key idea* behind this paper is to use code instrumentation methods from the domain of software profiling to estimate the task latencies and memory accesses accurately as shown in Figure 1. The framework uses the powerful source code instrumentation engine of a fine-grained software profiling tool to insert extra *function calls* inside software tasks. When the instrumented tasks are run inside the TLM environment, the extra function calls keep track of the cycle counts for the task execution, and intercept and forward the intra-task memory accesses to the TLM environment which simulates them over the communication infrastructure.

The automatically inserted *instrumentation code* provides an interface through which tasks can interact with the underlying communication network. Such interaction allows designers to consider performance and memory access bottlenecks of software execution while designing the commu-

3-9810801-0-6/DATE06 © 2006 EDAA

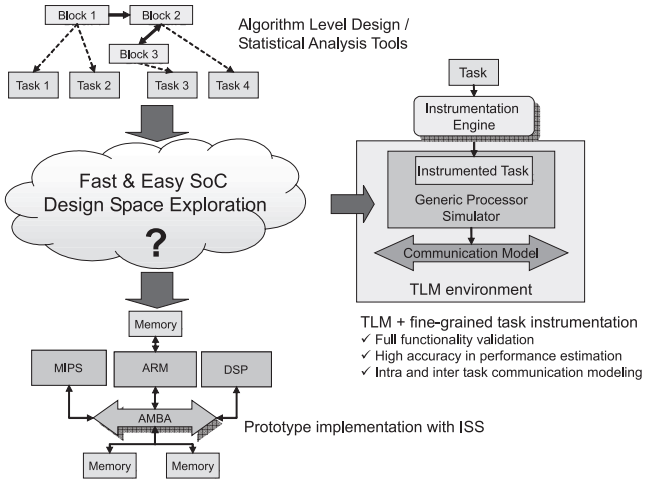


**Figure 1: Extended MP-SoC development**

nication architecture. Moreover, the network configurations and task structures are easy to change in the simulation environment, and task simulation is at least an order of magnitude faster than ISS. This results in fast refinement cycles with different network and task organizations, and rapid convergence to an optimal MP-SoC design.

After a discussion of the related work, we briefly present the methodology of the TLM environment and the software profiling tool used in our approach. In chapter 4 we introduce our concept and explain in detail the operational semantics of our framework, as well as the resulting design flow. Section 5 provides two case studies to demonstrate the capabilities of our methodology and the final section summarizes our work and presents some future directions.

## 2. RELATED WORK

Increasing complexity and heterogeneity of current and future MP-SoC architectures are forcing system architects to use new design techniques. System-Level-Design (SLD) is considered to be the appropriate way to cope with such rising complexity. Recently a number of different frameworks have been introduced to capture arbitrary Models of Computation (MoC) for the purpose of system level modelling and tooling [1, 2, 3]. The TLM framework used in this paper is based on SystemC due to its broad user acceptance and tool support.

The highest possible level of abstraction during design space exploration is based on statistical analysis [4] of the given application. Madsen et al. propose the combined modeling of NoC and RTOS scheduling at this high abstraction level, where the application tasks are only represented as a set of timing budgets for processing and communication without any functional information [5].

Otherwise complementary to our top-down refinement based design flow, the component design based paradigm [6] advocates bottom-up platform composition from a parameterizable IP library, containing common off-the-shelf (cots) platform elements. Such IP-based approaches bring clear advantages in rapid exploration and implementation of the general purpose and control-flow portion of the application, whereas our approach focuses on the execution of the data-plane processing.

In the field of simulation based SoC architecture analysis, the ARTEMIS [7] project is focused on automatic refinement of Kahn Process Network algorithm models to architecture models for the purpose of design space exploration and synthesis. The Modeling Environment for Software and Hardware (MESH) [8] is concerned with modeling heterogenous MP-SoC platforms above the cycle-level domain. Here schedulers are considered the central modeling element to capture the behavior of MP-SoC platform mappings.

Typically, the above mentioned SLD tools explore software execution either on high abstraction level by annotating estimated time consumptions, or evaluate the software's performance at low ISS level. Cai et al. propose a method for profiling and design space exploration of system architectures described in System-Level-Design Languages, such as SpecC or SystemC. However the storage accesses have to be explicitly modelled using predefined primitives in such frameworks. Our focus is on software profiling in SLD that does not require any explicit modeling of timing and/or memory accesses and is able to bridge the gap between abstract and fine-grained SW execution models. Therefore, the current work focusses on SW profiling and performance estimation at an *intermediate level* of abstraction.

Profiling, in itself, is a complex and well researched area, and is often used for *source code optimization* in the domain of general purpose computing. Tools such as GNU gprof/gcov [9] can provide users with per C statement or function level execution statistics for identification of program hot-spots. Unfortunately, such coarse grained information is not suitable for performance evaluation of embedded software. There also exists a number of assembly level profiling tools, such as the SpixTool [10] for SPARC, VTune [11] for Intel architectures or LISATek profiler [12] tied with LISA based Instruction Set Simulators (ISS), that can provide detailed processor specific information. However, such profilers are bound to specific architectures, and not suitable for performance estimation in a general, target processor independent way.

General, target independent, yet reasonably reliable software performance estimation methods have been proposed in [13, 14]. However, these tools do not provide estimates of memory accesses which are extremely important for MP-SoC design. [15] describes an approach for evaluating the performance and memory access patterns of multimedia applications through profiling. But this tool is conceived for algorithmic complexity evaluation, and its accuracy in performance estimation of embedded software has not been reported.

Tools such as [16] focus on estimating and counting memory accesses for power optimization, but ignore functional validation and verification.

The tool most suitable for our purpose of combining early SLD with fine-grained software execution is $\mu$P which has been designed for *pre-architecture* exploration for customizing or designing an Application Specific Instruction-set Processor (ASIP). As described in [17], the $\mu$P can provide fairly accurate performance and memory access estimates w.r.t. ISS and therefore, is the ideal vehicle for fast and early design space exploration.

## 3. DEVELOPMENT TOOLS

This section presents an overview of our TLM environment for early SLD space exploration. Then we introduce the Micro-Profiler ($\mu$P) [17] that allows early software code profiling and evaluation.

### 3.1 MP-SoC exploration framework

For early SLD space exploration we have already developed an MP-SoC exploration framework allowing simulation of a large number of Processing Elements (PEs) and di-

verse communication architectures, like Networks-on-Chip (NoCs). The framework is based on an extended y-chart principle [18], where architectural models, timing models and the functionalities of tasks can be independently developed. The communication protocols make use of a well defined TLM interface compliant to the OCP IP standard [19]. Using our framework, seamless migration from abstract to accurate models of both PEs and communication architectures can be performed through an iterative refinement process. Easy and fast refinement cycles are facilitated by XML based configurations allowing system modifications without the need of recompilation. A set of communication architectures (like the AMBA AHB bus) as well as a set of PEs (like the generic processor simulator called the *Virtual Processing Unit (VPU)* introduced in [20]) are already available within our framework.

Our framework is intended to be a workbench to assist the user in easy and fast evaluation of different system design decisions. It does not include automatic task partitioning, restructuring or software-hardware partitioning capabilities.

The major limitation of our current framework is the user-defined task timing model. The user has to manually precompute and annotate the timing information in task bodies. Such models often tend to be too optimistic, and mostly neglect memory accesses due to SW execution.

The key element of this paper is to cope with this issue and provide a fine-grained SW performance estimation. The $\mu$P, introduced in the next subsection, is the perfect tool to accomplish this goal.

## 3.2 Micro-Profiler

The micro-profiler ($\mu$P) is an application profiling tool designed to assist the designers of Application Specific Instruction-set Processors (ASIPs). Primarily, its goal is to bridge the gap between an application/algorithm (assumed to be given in C) and an ASIP architecture suitable to achieve the maximum performance for the target application. The $\mu$P provides the designer with important runtime statistics of the application, such as the usage statistics of different C operators for different data types, dynamic value ranges of data types and constants, coarse performance estimates etc., for an effective pre-architecture exploration to design or customize an ASIP.

The heart of $\mu$P is its fine-grained *code instrumentation* technique that inserts extra function calls in the original source of an application to collect pertinent runtime statistics. The next section shows how such code instrumentation can be used to connect software execution with a TLM environment to perform fast and effective design space exploration.

# 4. SW INSTRUMENTATION TECHNIQUE

In this section we present the key ideas behind our design flow and present the principle of our proposed work.

## 4.1 Key Ideas

As has been already mentioned in the introduction, the primary target of our work is to provide reliable software performance estimates in SLD that can be useful for making effective design decisions. This subsection briefly describes how and where we differ from existing SLD tools by highlighting the key aspects of the proposed design flow.

The key contribution of this paper is to apply *fine-grained, automatic source code instrumentation* techniques for implicitly modeling intra-task memory accesses and estimating

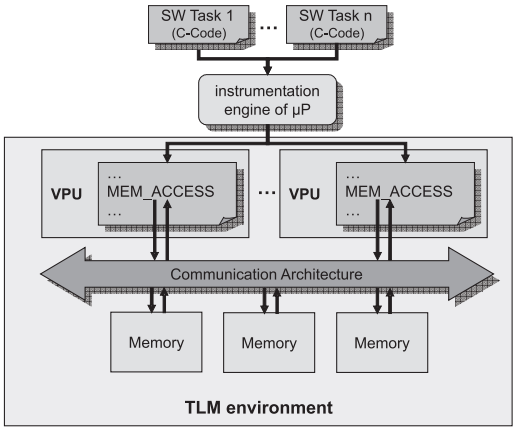


**Figure 2: Task execution work flow**

cycle counts. As shown in Figure 2, our instrumenter (based on the fine-grained $\mu$P tool) inserts additional *instrumentation code* into the tasks. Such instrumentation code dynamically increments cycle counters and redirects the intra-task memory accesses to the communication architecture when the corresponding task is executed on a VPU in our TLM environment.

Figure 3 on the next page depicts the common approaches to evaluate software code performance in today's SLD tools. At one extreme lies the *abstract* and *fast* statistical analysis tools where task models consist of only (statistically) estimated task timings. At the other end lies ISS based system modeling, which is accurate, but too slow and too involved for quick design space exploration.

As already mentioned in Section 3, [20] describes a task simulation framework on generic processor models called VPUs. One example of such a task is provided in Figure 3 as coarse-grained and manual instrumentation. This framework provides a way of validating task functionalities in early SLD, but the user still has to model the task timings explicitly. For example, in Figure 3, the user has to manually add the function call *consume* at the end of the task to increment the *cycle count* by a pre-computed value. Moreover, the intra-task memory accesses are difficult and tedious to model, and is therefore not usually considered.

The applied fine-grained software instrumentation, proposed within this paper, is done on *Three Address Code Intermediate Representation (3-AC IR)* level where all C operators and the majority of memory accesses are visible. Additionally, high level standard IR optimizations, such as constant propagation, constant folding and loop invariant code motion, can be performed on this IR. Such optimizations prevent chances of false prediction (such as counting operations that will be eventually eliminated by compiler optimizations) in estimating cycle counts and memory accesses. The accuracy of the software simulation is comparable to that of ISS, but it is almost an order of magnitude faster (as is shown by the experimental evidences presented later). The next subsections provide the details of our instrumentation framework and usage examples.

## 4.2 Instrumentation Principle

Since all operators and a majority of the memory accesses, are explicitly visible in 3-AC IR, our instrumenter, as shown in Figure 3, only needs to add an extra C line after each IR operation to increase the cycle counter by the *operator cost* (obtained through *GetOpCost*). The user can assign each

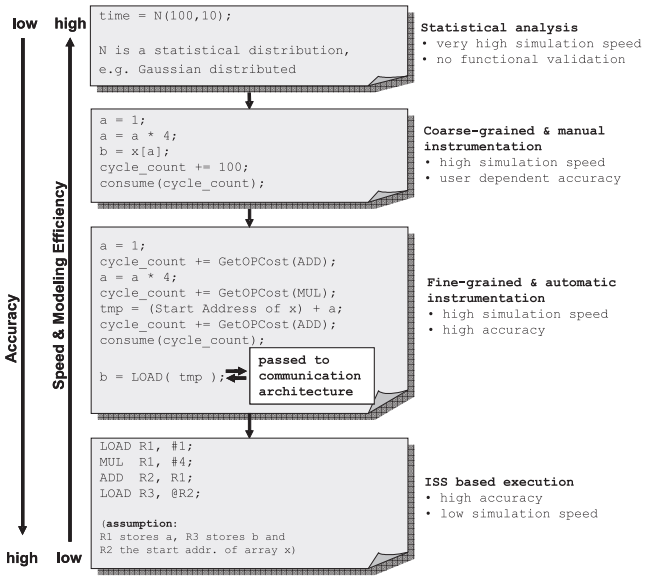


**Figure 3: SW performance estimation**

operator an appropriate cost that can be configured keeping the intended target processor in mind. For example, if the user intends to run a task on a processor which has a latency of three for multiplication and that of one for addition, then he can assign costs one and three to addition and multiplication operators, respectively. The total estimated execution time of the task is given by the following formula:

$$Cycles = \Sigma_{i=1}^{n} E(O_i) \times C(O_i)$$

where $E(O_i)$ and $C(O_i)$ are the execution count and cost for an operator $O_i$, respectively.

The instrumenter also inserts function calls to intercept and report all accesses to different source level data elements, such as arrays, structures and global variables, found in any application. Usually memory accesses, for an application written in a high level language like C, originate from four sources :

1. Accesses to global *scalar and composite variables* (i.e. structures) and arrays
2. Accesses to a function's local *composite* variables
3. Accesses to *dynamically allocated memory* on the *heap*
4. Accesses during building-up and cleaning-up of a function's stack frame

The local scalar variables are usually allocated in registers, and the number of memory accesses caused by them is often negligible. The 3-AC IR makes the first three kinds of memory accesses explicit by converting all global accesses and local composite accesses to pointer *dereference* operations.

An example is shown in Figure 3 where an array element is accessed by first calculating its address, and then loading it explicitly. The instrumenter only identifies this operation, and adds a function call that redirects this access to the communication network. After simulation of the memory access over the communication network, task execution continues performing other operations, and probably additional memory accesses.

One major limitation of our framework is that we cannot realistically estimate the memory accesses made in function prologues and epilogues. Therefore our approach can deviate significantly in predicting the total number of memory accesses when an application makes a considerable number of function calls. However we can statistically analyze the average number of memory accesses per function call for an application for a particular target processor architecture and increment the memory access counter accordingly.

## 4.3 Combined Execution

Figure 4 illustrates the execution of an instrumented task in our SLD framework. After instrumentation, the functionality of the task remains unchanged. However, after execution of each step, additional code (e.g. lines 3,5 and 7 in Figure 4) increases the cycle counter by the cost of the executed operation. If the user wants to roughly estimate the advantages/disadvantages of using one processor over another for the task execution, he needs only to change the costs of these operators in a configuration file. These *dynamically calculated* cycle counts are then returned as task latencies to the communication framework.

The simulation of intra-task memory accesses (as done by the *LOAD* function in line 9 of Figure 4) is a little more involved. The *LOAD* function is invoked with a *memory address*, and is expected to return the value contained in this memory address. At the same time, it is expected to simulate this memory access over the communication network and increment the cycle counter at the end of this simulation.

The problem, however, is that the task is compiled and executed on a host machine, and during execution, it accesses memory regions over which the simulation framework has no control. The instrumenter solves this problem by adding some initialization code that creates a map between the variable names and their runtime host machine addresses. When a memory access is made, functions such as *LOAD* can retrieve the name of a variable from this map based on the address supplied. It then consults another memory map that contains the *user defined memory location* for that variable in the *MP-SoC memory subsystem*. The communication network is simulated with this *SoC memory address* and the network activities and cycle count increases are duly noted.

For example, in Figure 4, the array $x$ can be mapped to an address $A_{host}$ in the host machine by the host compiler. At the same time, the user might want to see the effect of putting this element in address $A_{soc}$ (= 0x100) - some location residing inside, say, a shared memory accessed over a common bus. This can be done by adding an entry in the

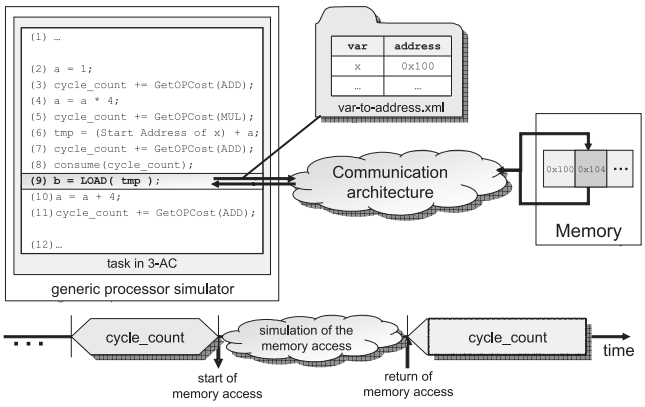


**Figure 4: Simulation environment**

configuration file. During execution, the LOAD function is called with the address of $A_{host}$ + a. LOAD first maps this address to the array variable $x$, and then simulates an access to the corresponding MP-SoC memory region starting with $A_{soc}$.

Such configurable memory mapping enables the user to explore different memory architectures very easily. For example, in Figure 4, the user can experiment with the performance of the communication architecture, its load characteristics, the overall memory access latencies and the resulting slowdown/speed-up of the task, by putting the variable $x$ in shared or dedicated memories, accessed over different communication architectures etc. by editing a few XML files.

# 5. CASE STUDY

This section presents case studies to demonstrate the possible uses of our techniques in SLD. Any SLD tool for *rapid* and *early* design space exploration must be benchmarked with the following criteria in mind

1. **Speed and modeling efficiency**. Tools for early design space exploration must have a clear advantage, in terms of speed and ease of applying model changes, over techniques suitable for later design phases.

2. **Accuracy** of the results. Although at early stages of design it is difficult to quantify the exact performance benefits of various design choices, the *relative* merits/demerits of different decisions must be clearly visible.

3. **Usefulness** that is characterized by the kind of information/design hints the user can obtain from the tool and how easily that can be translated to a design decision.

The following two subsections provide two case studies to rate the current work w.r.t. the above parameters. The final subsection summarizes the results of the case studies.

The applications we have selected for the case studies come from *network processing* and *speech processing* - two of the most prominent embedded application domains. The first algorithm is the well known *blowfish* symmetric key block cipher algorithm. The second one is *G.729* - an advanced speed processing algorithm. The SoC architecture chosen for the case study consists of a single MIPS32 [21] processor core connected to a flat memory over a single AMBA AHB bus.

Each application has been instrumented through $\mu$P and converted to tasks that can be run on our framework. The tasks have been run on a VPU parameterized according to the MIPS32 characteristics. We compared the results obtained through our framework to a reference system running one LISATek based MIPS32 *Instruction Accurate* ISS and having the same memory architecture. Since our main goal is to demonstrate the usage scenarios of our tools, we only suggest hints for improving the communication architecture.

## 5.1 Blowfish

The *Blowfish* algorithm, designed by B. Schneier, [22] is a symmetric block cipher algorithm with 64-bit block size and variable length keys (up to 448 bits). Our implementation of the algorithm consists of three different functions initialization, encryption and decryption. The accuracy comparison of the instrumented code for each function w.r.t. that of MIPS ISS is summarized in Figure 5. The instrumented code running on the VPU deviates only by 8% and 14%

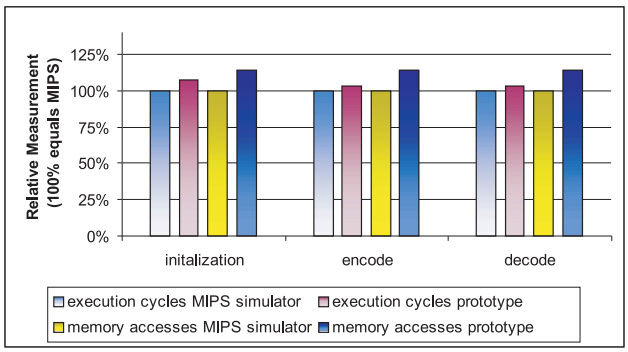


**Figure 5: Results of the blowfish algorithm**

in predicting the execution cycles and memory accesses, respectively. Such estimates fall well within any reasonable limit of tolerance for early design space exploration. The simulation speed is at approximately *2.3 Mcycles/sec*, which is a factor of *9.1* faster compared to instruction accurate ISS based execution.

A closer look at the simulation results reveals that more than 80% of the intra-task memory accesses go to array variables commonly known as *S-boxes*. Such a biased memory access pattern dictates that these S-boxes should be moved to a fast memory (such as a scratch-pad) connected over a local bus to the processor to reduce memory latencies and power consumption. Following this hint we added an exclusive memory and mapped the S-Boxes into that. Then we performed simulations of the application with different memory access latencies. The results are illustrated in Table 1 which shows that the decryption algorithm can have significant speed-up by merely moving the S-Boxes to the exclusive memory. Such changes to the architecture require modifications of less than 10 lines in the XML configuration files and therefore can be performed in a couple of minutes. This demonstrates the fidelity of our proposed framework in assisting the system architect to perform simulation, modification, validation and exploration of design decisions quickly.

| memory latency in cycles (for S-Box variables) | execution time of one decryption operation |
|---|---|
| 10 | 100% |
| 5 | 71.5% |
| 3 | 62.2% |
| 1 | 57.4% |

**Table 1: Effect of S-Box mapping**

## 5.2 G.729

The *G.729* algorithm [23] is a fairly complex 8-kbps Conjugate-Structure Algebraic-Code-Excited Linear Prediction (CS-ACELP) speech compression algorithm. This audio data compression algorithm is standardized by the International Telecommunication Union (ITU) and is mostly used in Voice-over-IP (VoIP) products. The speed and cycle count comparisons for this algorithm w.r.t. MIPS32 ISS are shown in Figure 6. The execution cycle estimates are off by approximately 7.5%, whereas the deviation in memory access estimates is slightly higher at 20%. This relatively high (than blowfish) error in memory access profiling results due to large chain of function calls, and hence large number of loads and stores in function prologues and epilogues,

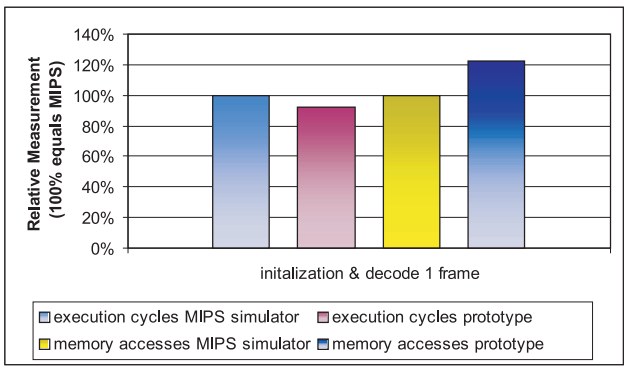


**Figure 6: Results of the G.729 case-study**

which is difficult to estimate at 3-AC IR level. The simulation speed is again increased by a factor of approximately *9* compared to instruction accurate ISS based execution.

A deeper look at the memory access behavior of the application showed the following characteristics

- 40% of all memory accesses occur in reading and writing the input and output data streams, respectively.
- During the frame decode operation, generation of the excitement vector [23] causes approximately 20% of all memory accesses.

Following these hints, the user might be tempted to put the input and output data-streams and the excitement vector arrays in fast and local memories. The impact of this decision on the overall system performance can be easily evaluated by changing the network configuration files and the corresponding memory maps.

## 5.3 Result Summary

This subsection summarizes the results obtained through our tools. The tools can achieve from 80% to 98% accuracy in cycle count estimates and memory profiles. Moreover, the speed of profiling is at least one order of magnitude faster than that of ISS. This makes it ideal for early design space exploration.

Our framework permits a variety of experiments with the software and the communication architecture. For example, the user can see the effects of diverse memory architectures by simply changing the memory map configurations files, or he can evaluate the results of having a different processor core by simply changing the costs of different operators.

Our tools provide an excellent framework for Software/SoC co-simulation from very early stages of design. The first advantage is that the software development can start right from the beginning of system design and does not need to wait till the initial architecture is finalized. This can result in significantly shorter design cycles. Moreover, hints from software evaluation can be easily incorporated into hardware development and its effects seen from early design stages. Such synergetic development is a must for future MP-SoCs.

# 6. CONCLUSION

In this paper, we propose an approach that assists simultaneous software and hardware development, functional validation and evaluation at SLD. The main contribution of this paper lies in combining two different worlds — early task level MP-SoC architecture exploration, and software profiling for performance estimation — into a single framework. The combined tool-set allows easy and fast design space exploration with high simulation speed and modeling efficiency, and enables system architects to design and adapt SoC communication architectures according to the requirements of software. The combined software-hardware simulation facilitates a stepwise refinement of both the software task model and the SoC architecture from the very beginning of the design cycle. Two case studies show the efficiency of the design flow, and demonstrate how the tool-set can be used in an iterative refinement process.

The major shortcoming of our approach is the inability of our software profiler to provide reliable cycle count estimates for non-RISC architectures, especially VLIW and super-scalar machines. Our future work will mostly concentrate on this issue. Taking effects of Operating Systems (OS) and Real-Time services into our performance estimation is another promising area of work.

# 7. REFERENCES


[1] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[3] D. Gajski, J. Zhu, R. Dömer et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[4] L. Thiele, S. Chakraborty, M. Gries, S. Kunzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proc. of the Design Automation Conference (DAC)*, 2002.

[5] Jan Madsen et al. Network-on-chip modeling for system-level multiprocessor simulation. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium RTSS03*, pages 82–92, December 2003.

[6] M.-A. Dziri, W. Cesio, F.R. Wagner, A.A. Jerraya. Unified Component Integration Flow for Multi-Processor SoC Design and Validation. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2004.

[7] A.D. Pimentel, L.O. Hertzberger, P. Lieverse, P. van der Wolf, E.F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[8] J.M. Paul, A. Bobrek, J.E. Nelson, J.J. Pieper, D.E. Thomas. Schedulers as Model-Based Design Elements in Programmable Heterogeneous Multiprocessors. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[9] GNU. *http://www.gnu.org/*.

[10] Sun Microsystems. *SpixTools: Introduction and User's Manual, TR-93-6*.

[11] Intel. *VTune*.

[12] LISATek Product Line. *CoWare, http://www.coware.com*.

[13] E. Harcourt P. Giusto, G. Martin. Reliable estimation of execution time of embedded software. In *"Proc. Int. Conf. on Design, Automation and Test in Europe(DATE)"*, 2001.

[14] E. Harcourt et al L. Lavagno, J. R. Bammi. Software performance estimation strategies in a system-level design tool. In *"Proc. Int. Symp. on Hardware/Software Codesign (CODES)"*, 2000.

[15] M. Mattavelli M. Ravasi. High-level algorithmic complexity evaluation for system design. In *Journal of Systems Architecture, no. 48*, Elsevier, 2003.

[16] Power Escape. *http://www.powerescape.com/*.

[17] K. Karuri et al. Fine-grained Application Source Code Profiling for ASIP Design. In *42nd Design Automation Conference*, Anaheim, California, USA, June 2005.

[18] E. Bensoudane P.G. Paulin, C. Pilkington. Stepnp: A system-level exploration platform for network processors. *IEEE Design & Test of Computers*, 19(6):17–26, Nov-Dec 2002.

[19] OCP IP. *http://www.ocpip.org/*.

[20] T. Kempf, T. Kogel et al. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *Proc. of the Conf. on Design, Automation & Test in Europe(DATE)*, Munich, Germany, March 2005.

[21] MIPS Technologies Inc. *http://www.mips.com/*.

[22] B. Schneier. *Applied Cryptography*. Addison-Wesley Publishing Company, Boston, June 1996.

[23] Recommendation G.729. *http://www.itu.int/*.