# DSP Processor/Compiler Co-Design: A Quantitative Approach

Vojin Živojnović, Stefan Pees, Christian Schläger,
Markus Willems, Rainer Schoenen and Heinrich Meyr
Aachen University of Technology
Integrated Systems for Signal Processing
Templergraben 55, IS2 611810, 52056 Aachen, Germany
zivojnov@ert.rwth-aachen.de

## Abstract

*In the paper the problem of processor/compiler co-design for digital signal processing and embedded systems is discussed. The main principle we follow is the top-down approach characterized by extensive simulation and quantitative performance evaluation of processor and compiler. Although well established in the design of state-of-the-art general purpose processors and compilers, this approach is rarely followed by leading producers of signal and embedded processors. As a consequence, the matching between the processor and the compiler is low. In the paper we focus on three main components of our exploration environment — benchmarking methodology (DSPstone), fast processor simulation (SuperSim), and machine description (LISA). Most of the paper is devoted to the technique of compiled processor simulation. The speedup obtained allows an exploration of a much larger design space than it was possible with standard processor simulators.*

## 1. Introduction

Compiler efficiency is currently not the main feature guiding processor selection for some digital signal processing (DSP) application. However, the constantly tightening time-to-market constraints, falling hardware prices, and new faster VLSI technology could promote compiler efficiency to a factor of highest importance very soon. The current approach to the design of DSP processors and compilers is characterized by a sequential design flow. First, the DSP processor is completely developed. Thereby, the guiding design criterions — chip area and power consumption — are iterated to meet performance on some selected application kernel, like e.g. FIR filter. Having the DSP processor chip in their hand DSP compiler designers try to map the compiler to the processor in the best possible way. The result is mostly a suboptimal solution in which costly resources in terms of chip area and design effort cannot be accessed by the compiler, and in the same time resources important for compiler performance are missing.

The goal of this paper is to present the main ideas and design steps of the quantitative approach to processor/compiler co-design. In order to support the proposed methodology we concentrate on three main issues — performance measurement and evaluation, fast processor simulation, and formal machine specification. At our Institute these issues are currently covered by three projects — DSPstone, SuperSim, and LISA respectively. In the paper we provide an update on these projects, and explain their contribution to the proposed quantitative design methodology.

## 2. Quantitative Approach to DSP Processor/Compiler Co-Design

Joint development of processor and compiler is a well known approach to the design of general purpose computers. It started with the introduction of RISC architectures in early-80s [7], and continued in mid-80s as the main strategy for high performance processor/compiler solutions (see e.g. [12]). Thereby, quantitative performance measurements evolved as the primary guiding force of the design process [6,5,10].

In the field of DSP processors the old-style development fashion — compiler-after-processor — was followed for a long time. Although new compiler-friendly DSP processors were announced recently (e.g. ADSP21csp01 of Analog Devices), the general retard in

design methodology is still large. It has two main causes. First, until recently the volume of the DSP market was much smaller than the volume of the GPP market. As a consequence, the pressure to resort to more efficient design methodologies was lower. As second, with the increase in complexity of DSP applications in the wireless and multimedia fields the role of the compiler, and the interplay between compiler and processor became more important.

The joint processor/compiler design flow is presented on Figure 1. Processor and compiler development

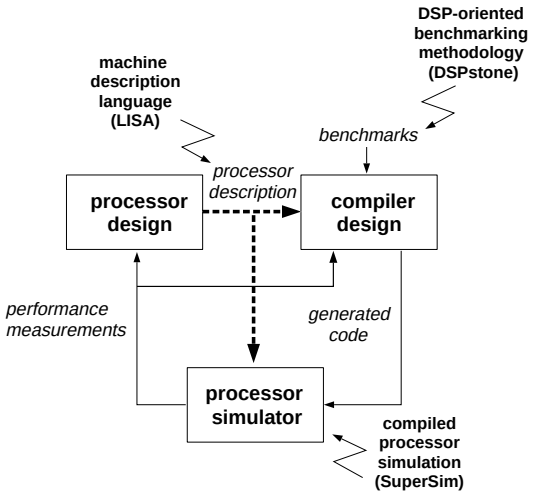


**Figure 1. Quantitative Approach to Processor/Compiler Co-Design.**

are governed by performance measurements delivered by the simulator of the processor. After processor redesigns, the updated processor description is used to retarget the compiler. The design cycle is closed by compilation of the benchmarks and feeding the simulator with the generated assembly code.

Although this general strategy can be applied to DSP processor/compiler development equally well, peculiarities of DSP applications, architectures, and performance criterions require a specific, DSP-oriented approach. In order to build the necessary tool-base for joint DSP processor/compiler development, we concentrated on specific, DSP-related requirements on benchmarking, processor simulation, and machine description formalisms. We covered these three issues with three projects — DSPstone, SuperSim, and LISA (Fig. 1). In the next sections we shall report the main results.

## 3. DSP-Oriented Benchmarking Methodology - DSPstone

The DSPstone project started in 1993 [14]. The initial goal was to provide quantitative efficiency measures of commercially available DSP compilers, and replace thereby the unreliable qualitative measures provided by producers of DSP compilers. Using DSPstone the potential compiler user can decide whether a DSP compiler represents a reasonable alternative to manual assembly coding of his application. During time DSPstone became increasingly popular among compiler and processor producers. Currently it is used for design tuning and testing by different companies and research groups.

DSPstone consists of three benchmark suites (application, DSP-kernel, and HLL-kernel suite) with over 20 benchmarks, and a DSP-oriented measurement methodology. Benchmarks are typical DSP applications (e.g. ADPCM transcoder), DSP-specific code kernels (e.g. FIR filter), or code fragments specific for high-level languages (e.g. function calls). The methodology compares the code generated by the compiler to the hand written assembly reference code in three dimensions — execution time, program, and data memory size. The assembly reference code is assumed to be the optimal code which can be written for a particular processor, and is mostly taken from vendor supplied assembly libraries. The possibility to measure the code against a fixed reference is the unique feature of DSPstone. For standard general purpose processing (GPP) benchmarks, like SPEC or Dhrystone, functionally equivalent assembly versions of the benchmarks cannot be found, and the reference code method is not applicable.

As an example, Figure 2 presents the benchmarking results of the STARTUP application benchmark for four fixed-point DSP processors, and a single GPP processor. The STARTUP benchmark is the startup synchronization protocol of the ITU-V.32 modem standard. DSPstone results show that all DSP compilers introduce a significant overhead in execution time and program code. In order to explore the behavior of GPP compilers we applied the DSPstone methodology to the Pentium processor with the Borland compiler. Our conclusion is that this well known GPP compiler introduces a high overhead, too. However, this overhead is mostly tolerated because of the application domain, high complexity of GPP applications, and GPP-specific performance criterions.

Our next example is the ADPCM application benchmark, which is a full implementation of the ITU-G.721 speech compression standard. In this case we obtained

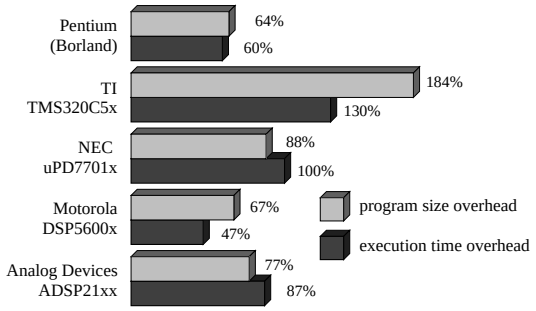


**Figure 2. STARTUP Benchmark: Program Size and Execution Time Overhead.**

execution time overheads between 500% and 700%. To provide a better insight into the behavior of DSP compilers and their interaction with the architecture, we analyzed the dynamic instruction distribution (DID) of the ADPCM code. These distributions show the execution frequencies of instructions from specified instruction classes. Figure 3 shows the DID of the assembly reference code and the DID of the compiled code for the Motorola DSP5600x compiler. Three instruction classes were defined — move, control, and arithmetic-logic (ALU). The difference in distributions between

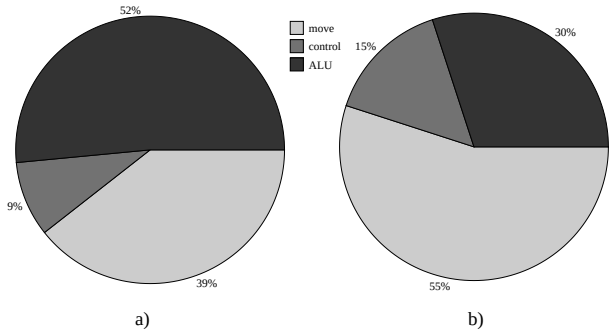


**Figure 3. ADPCM Benchmark for DSP 5600x: DID of a)handwritten, b)compiled.**

the compiled and handwritten code for some instruction class indicates that the inefficiency of the compiler does not influence all instruction classes equally. The percentage of move and control instructions is much higher for the compiled than for the handwritten assembly code. We have concluded that one of the reasons for this behavior lies in the compilation technique itself. The intermediate representation of the C code is translated into fragments of assembly instructions which are glued together using many register-to-register and memory-to-register move instructions.

Examples provided above show that the quantitative approach to compiler evaluation can deliver useful indicators of compiler efficiency. Although the current version of DSPstone provides only the overhead measures, like those for the STARTUP benchmark, we believe that the dynamic instruction distribution can be an additional aid during the compiler design, testing and evaluation process.

## 4. Compiled Processor Simulation - SuperSim

Interpretive simulators process instructions using a software model of the target processor. A virtual processor model is built using a data structure representing the state of the processor and a program which changes the processor state according to the stimuli — either a new instruction pointed to by the program sequencer, or some external events, such as interrupts. In general, interpretive simulators can be summarized as a loop in which instructions are fetched, decoded and executed using a "big switch" statement. All currently available instruction set simulators of DSP processors use the interpretive simulation technique. Their main disadvantage is the low simulation speed (2K-20K insns/s [11]). On the other hand, any type of dedicated hardware realization, like emulation, would increase the design cycle enormously.

Compiled processor simulation we use enables speedups of up to three orders of magnitude. It follows the same main idea found in compiled simulation of synchronous VLSI circuits [13], constant propagation in HLL compilers [1], or static scheduling of [4]. The idea is to use a priori or compile-time knowledge in order to reduce run-time computation. It is a typical application of the principle "make frequent operations fast". Our approach resembles binary translation used for migrating executables from one machine to another [8,10], or collecting run-time statistics [3]. However, clock/bit-true translation and debugging are not objectives of binary translation.

Let us assume a state-space model of the processor. The machine state is changed by applying the transition functions which are selected by the transition control. Figure 5 shows the interaction between the machine state, transition control and transition function. The machine state consists of the control state (instruction memory and control registers) and the data state (data memory and data registers). In most programs of practical interest the update of the control state by the transition function (dotted on Figure 5) is less frequent than the update of the data state. Therefore, we can assume that for most of the time the transition con-

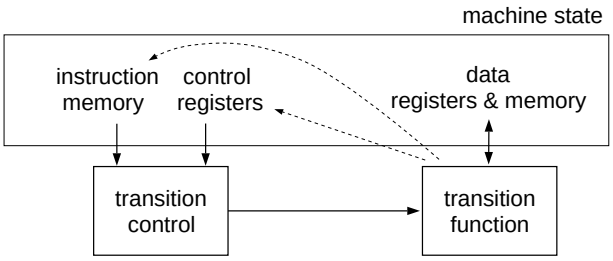


**Figure 4. Machine State, Transition Function, and Transition Control.**

trol operates on constants, and correct the transition control only if updates of the control state happen. If the speedup obtained by pre-compiling the transition control is greater than the slowdown introduced by the necessary correcting actions, then an overall simulation speedup is obtained.

Programs updating the instruction memory are known as self-modifying programs, and are found quite rarely in the programming practice. In digital signal processors and microcontrollers self-modifying code is found mostly in form of changing the target address field of a jump instruction. However, this type of self-modifying code can be handled at run-time without any penalty.

The correction of the transition control is necessary for instructions altering the control registers, like program control instructions (e.g. jumps and calls) and interrupts. The transition control has to be supplied with appropriate run-time instruments to handle these conditions. The necessary run-time overhead ranges from simple (non-pipelined processors) over moderate (pipelined processors) to complex (processors with out-of-order execution).

In processor simulation there are not only two extremes — interpretive and compiled computation of the transition control, but a number of possible compile-/run-time trade-offs which offer different simulation performance. Three main levels of compiled simulation can be distinguished.

*Compile–Time Decoding (Table–Driven Simulation)* – If we assume that the instruction memory is not altered during run-time, the decoding operation can be done completely at compile-time. The amount of speedup depends on the type of target instruction encoding and the relative frequency of the write access to program memory. During instruction fetch the instruction table is accessed instead of the instruction memory. In this way the target machine is converted into a functionally equivalent machine with a less compact instruction coding scheme. The correcting action is necessary only if the code is of self-modifying type. In this case each write access to the instruction memory has to be followed by an update of the instruction table. In most cases of practical interest this happens less frequently than the decoding operation, and the overall speedup is significant.

*Compile–Time Instruction Sequencing (Unfolding of the Simulation Loop)* – In standard interpretive simulators the simulation loop is iterated over instructions of the program. We can unfold the simulation loop by an unfolding factor equal to the number of instructions in the program. The advantages of simulation loop unfolding is in the reduction of the looping overhead. The price is paid in increased program memory utilization on the host side, and a possible slowdown for hosts supporting caching. However, for targets like DSPs and microcontrollers this should not be a problem — the program memory is mostly much smaller than the host cache, and the locality of reference of the target program very high.

*Compile–Time Instantiation (Binary–to–Binary Translation)* – Compile–time instantiation combines the previous two techniques into one. Under the assumption that both the instruction memory and the control registers do not change, each of the simulation calls of the unfolded simulation loop can be parameterized by the corresponding transition function and operands, and instantiated already at compile-time. In this way the binary code of the target is translated into the binary code of the host. For instructions affecting the control state of the processor, a correcting action which changes the control flow of the simulation program is necessary.

Depending on the processor architecture, the three levels of compiled simulation can have different influence on the performance. However, even for a single simulation it is possible to switch from one level to the other depending on the control flow of the processor. For example, pipelines can be modeled using binary-to-binary translation until indirect branches or interrupts are encountered. At that moment we have to switch to the table-driven execution and instantiate the simulation code at run-time. As soon as there are no indirect branches or interrupts in the pipeline, we can switch back to the translated simulation code. A detailed analysis of compiled simulation of pipelines is provided in [15].

### 4.1. An Example: Compiled Processor Simulator for the ADSP21xx DSP

The *SuperSim* SS-21xx compiled simulator has been implemented for the Analog Devices ADSP-21xx fa-

mily of DSP processors. It provides bit- and clock accurate simulation of the core and the peripherals. Table 1 presents some real-life examples of SS-21xx performance. Simulation speed measured in insns/s depends on the complexity of instructions found in the target code. The FIR filter example is generated by the C compiler of the target that generates multi-operation instructions rarely. However, the ADPCM example is hand-coded optimally and uses multi-operation instructions frequently. The results from Table 1 show

| example | simulator | opt. | insns/s | speedup |
|---|---|---|---|---|
| FIR filter | ADSP-21xx | - | 3.9K | 1 |
| | SS-21xx | -O3 | 2.5M | 640 |
| | -"- | -O2 | 2.0M | 510 |
| | -"- | -O1 | 1.6M | 420 |
| | TI-C50 | - | 2.4K | 1 |
| | SS-C50† | -O3 | 0.8M | 320 |
| | SS-77016† | -O3 | 0.8M | - |
| ADPCM | ADSP-21xx | - | 4.0K | 1 |
| | SS-21xx | -O3 | 0.8M | 200 |
| | -"- | -O2 | 0.6M | 150 |
| | -"- | -O1 | 0.4M | 100 |
| host: Sun-10/64MB; SS-21xx flags: -f compiler: gcc 2.5.8; †-preliminary results; | | | | |

**Table 1. SS-21xx Performance.**

that our simulator outperforms the standard simulator by almost three orders of magnitude on the FIR example and by about 200 times on the ADPCM example. The same verification which took 6.4 hours with the standard ADSP-21xx simulator is reduced to less than 2 minutes using SS-21xx.

The ADSP-21xx processor does not have a visible pipeline. In order to prove our concepts on architectures with pipeline effects we have written compiled simulation examples for the TI's TMS320C50 and NEC's $\mu$PD77016 processors. Despite of the overhead introduced by pipeline modeling, results from Table 1 show that our approach still achieves a significant speedup.

## 5. Machine Description Language - LISA

LISA is a machine description language developed to parameterize a generic machine model, and in this way deliver the final machine model which is used for simulation and compilation. The goal is to have a single generic machine model, and a single machine description language covering a whole class of processor architectures (Figure 6).

Our wish to capture a more detailed timing information, than that of ISA, motivated the introduction of operation-level behavior and scheduling descriptions.

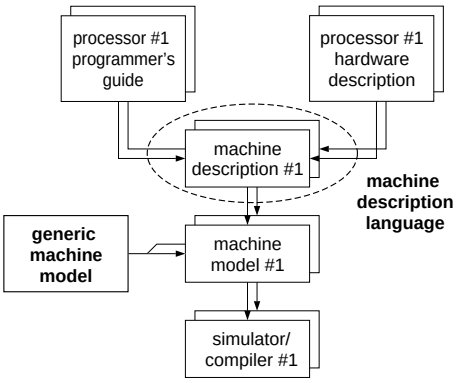


**Figure 5. LISA - Machine Description Language and Generic Machine Model.**

In terms of modeling the operation sequencer, LISA follows the same main idea of reservation tables and Gantt charts, like in [2] and [9]. However, in order to enable modeling of data/control hazards, and pipeline flushes, we extended the modeling ability of Gantt charts by introducing operation descriptors.

To enable clock-accurate modeling, instructions are partitioned into operations, as basic schedulable units. At each clock-cycle the admissible operations are combined into a single transition function which changes the machine state. Admissible operations are determined from precedence and resource constraints specified for operations of each instruction. Figure 7 shows how the information from the Gantt charts is used by the operation sequencer in order to determine admissible operations at each control step. In our case the time

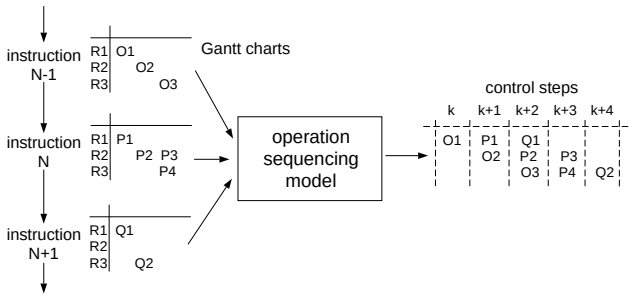


**Figure 6. Gantt Chart Scheduling.**

axis of the Gantt chart is converted into a precedence axis, so that precedence and resource constraints are specified. The sequencer determines admissible operations according to the As Soon As Possible (ASAP) principle, thereby accounting for precedence and resource constraints imposed by the Gantt charts of instructions. If multiple operations coming from different

instructions compete for the same resource, the precedence is determined by the (logical) precedence of their instructions as specified by the program thread.

The proposed generic machine model is well suited to model statically scheduled pipelines. We assume that the pipeline fetches an instruction and issues it, unless there is a conflict with previous instructions. Dynamic pipeline sequencers, where the hardware rearranges the instruction execution to reduce the stalls, still have to be explored.

Currently, the primary application domain of LISA is timed ISA simulation (LISA/S). We hope to present the work on the LISA compilation model (LISA/C) soon.

## 6. Conclusions and Future Work

The DSPstone benchmarks and the evaluation methodology provide quantitative guidance of the design process. We have shown that quantitative analysis is not restricted only to benchmark definition and overhead measures of execution time and memory utilization. The dynamic instruction distributions we compute provide additional information about the performance of the joint processor/compiler system. We found DIDs especially useful for tuning compiler optimizations and processor characteristics with contradicting effects.

During processor/compiler co-design the processor is in development, so simulators are the only way to estimate performance. Reliable performance indicators are obtained only if complete real-life applications with large number of instructions are simulated. As a consequence, the simulator becomes the bottleneck of the design procedure. We propose the usage of compiled simulators which deliver a much higher simulation speed than the standard interpretive simulators. Our experiments with the compiled simulator for the ADSP21xx signal processor have shown speedups up to three orders of magnitude. We expect that the need for fast and easy retargetable simulators of DSP and embedded processors will grow in the future.

In order to automate the retargeting of simulator and compiler after processor redesigns, we investigated machine description languages and defined the new machine description language LISA. The novelty of LISA is in the generic machine model and its operation-based sequencing model. The new language and the accompanying generic machine model are able to capture all effects of pipelines found in DSP and embedded processor.

Our future work shall focus on a GUI interface for LISA programming, and a tool for power and area estimation of the explored architecture. In this way the processor design can be additionally guided by implementation characteristics of the final hardware, so that implementation-related corrections of the architecture can be done already at an early design stage. Our ultimate goal is a complete DSP processor/compiler co-design environment. Similar environments already exist for exploration of GPP processors and compilers. Using their experience we shall focus on applications and requirements which are specific for DSP and embedded systems.

## References


[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] D. Bradlee, R. Henry, and S. Eggers. The Marion system for retargetable instruction scheduling. In *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada*, pages 229–240, 1991.

[3] J. Davidson and D. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15:459–472, Nov. 1991.

[4] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, C-36:24–35, Jan. 1987.

[5] J. Hennessy and M. Heinrich. Hardware/software codesign of processors: concepts and examples. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design.* Kluwer Academic Publishers, 1995.

[6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[7] D. Patterson and C. Séquin. A VLSI RISC. *IEEE Computers*, pages 8–21, Sep. 1982.

[8] R. Sites, *et al.* Binary translation. *Comm. of the ACM*, 36:69–81, Feb. 1993.

[9] B. Rau. VLIW compilation driven by a machine description database. In *Proc. 2nd Code Generation Workshop, Leuven, Belgium*, 1996.

[10] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, pages 34–43, Winter 1995.

[11] J. Rowson. Hardware/Software co-simulation. In *31st ACM/IEEE Design Automation Conference*, 1994.

[12] M. Tremblay and P. Tirumalai. Partners in platform design. *IEEE Spectrum*, pages 20–26, Apr. 1995.

[13] Z. Barzilai, *et al.* HSS - A high speed simulator. *IEEE Trans. on CAD*, CAD-6:601–616, July 1987. 1987.

[14] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proc. of ICSPAT'94 - Dallas*, Oct. 1994.

[15] V. Živojnović, S. Tjiang, and H. Meyr. Compiled simulation of programmable DSP architectures. In *Proc. of IEEE Workshop on VLSI in Signal Processing, Osaka, Japan*, pages 187–196, Oct. 1995.