Sandbridge Software Tools

John Glossner, Sean Dorward, Sanjay Jinturkar, Mayan Moudgill, Erdem Hokenek, Michael Schulte, and Stamatis Vassiliadis

***Abstract*—We describe the generation of the simulation environment for the Sandbridge Sandblaster multithreaded processor. The processor model is described using the Sandblaster architecture Description Language (SaDL), which is implemented as python objects. Specific processor implementations of the simulation environment are generated by programmatically calling the python objects. Using just-in-time compiler technology, we dynamically compile an executing program and processor model to a target platform providing fast interactive responses with accelerated simulation capability. Using this approach, we simulate up to 100 million instructions per second on a 1 GHz Pentium processor. This allows the system programmer to prototype many applications in real-time within the simulation environment, providing a dramatic increase in productivity and allowing flexible hardware-software trade-offs.**

***Index Terms* — Architecture Description Language, Multithreaded Architecture, Compiler, Simulation, Compiled Simulation.**

I. Introduction

THE *architecture* of a computer system is the minimal set of properties that determine what programs will run and what results they will produce [1]. It is the contract between the programmer and the hardware. Every computer is an interpreter of its *machine language* – that representation of programs that resides in memory and is interpreted (executed) directly by the (host) hardware. A *simulator* is an interpreter of a machine language where the representation of programs resides in memory but is not directly executed by host hardware. Historically, three types of architectural simulators have been identified. An *interpreter* consists of a program executing on a computer, where each machine language instruction is executed on a model of a target architecture running on the host computer. Because interpreted simulators tend to execute slowly, compiled simulators have been developed. A *statically compiled simulator* first translates both the program and the architecture model into the host computer's machine language. A *dynamically compiled* (or just-in-time) *simulator* either starts execution as an interpreter, but judiciously chooses functions that may be translated during execution into a directly executable host program, or begins by translating at the start of the host execution.

A. Interpreted Execution

Instructions set simulators commonly used for application code development are cycle-count accurate in nature. They use an architecture description of the underlying processor and provide close to accurate cycle counts, but typically do not model external memories, peripherals, or asynchronous interrupts. However, the information provided by them is generally sufficient to develop the prototype application.

Figure 1 shows an interpreted simulation system. Executable code is generated for a target platform. During the execution phase, a software interpreter running on the host interprets (simulates) the target platform executable. The simulator models the target architecture, may mimic the implementation pipeline, and has data structures to reflect the machine resources such as registers. The simulator contains a main driver loop, which performs the *fetch, decode, data read, execute and write back* operations for each instruction in the target executable code.

An interpreted simulator has performance limitations. Actions such as instruction fetch, decode, and operand fetch are repeated for every execution of the target instruction. The instruction decode is implemented with a number of conditional statements within the main driver loop of the simulator. This adds significant overhead especially considering all combinations of opcodes and operands must be distinguished. In addition, the execution of the target instruction requires the update of several data structures that mimic the target resources, such as registers, in the simulator.

Dr. John Glossner is CTO & Executive Vice President, Sandbridge Technologies, 1 North Lexington Ave., White Plains, NY, 10512, USA. (phone: 914-287-8500; fax: 914-287-8501; e-mail: jglossner@sandbridgetech.com).

Sean Dorward is a Principal Engineer at Sandbridge Technologies, 1 North Lexington Ave., White Plains, NY, 10512, USA. (phone: 914-287-8500; fax: 914-287-8501; e-mail: sdorward@sandbridgetech.com).

Dr. Sanjay Jinturkar is Director of Software, Sandbridge Technologies, 1 North Lexington Ave., White Plains, NY, 10512, USA. (phone: 914-287-8500; fax: 914-287-8501; e-mail: sjinturkar@sandbridgetech.com).

Dr. Erdem Hokenek is Chief Hardware Architect at Sandbridge Technologies, 1 North Lexington Ave., White Plains, NY, 10512, USA. (e-mail: hokenek@sandbridgetech.com).

Dr. Mayan Moudgill is Chief Software Architect at Sandbridge Technologies, 1 North Lexington Ave., White Plains, NY, 10512, USA. (e-mail: mayan@sandbridgetech.com).

Dr. Michael Schulte is an Assistant Professor with the Electrical and Computer Engineering Department at the University of Wisconson-Madison. 1415 Engineering Drive, Madison, WI 53706, USA (e-mail: schulte@engr.wisc.edu).

Prof. Dr. Stamatis Vassiliadis is a Chair Professor in the Electrical Engineering Department at TU Delft, Mekelweg 4, 2628 CD Delft, The Netherlands. (e-mail: s.vassiliadis@its.tudelft.nl).

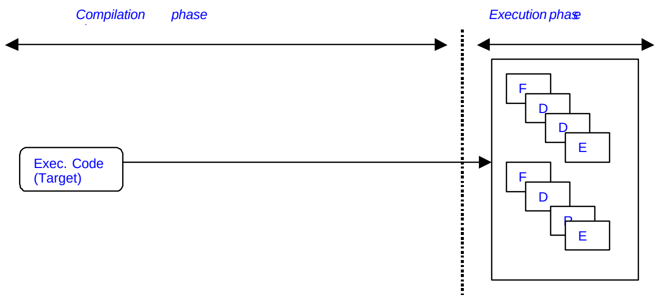

Figure 1. Interpreted Simulation

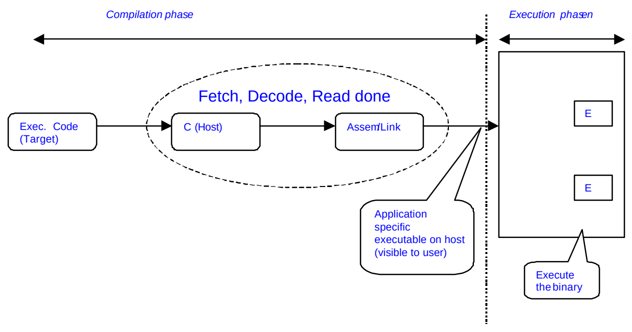

Figure 2. Statically Compiled Simulation

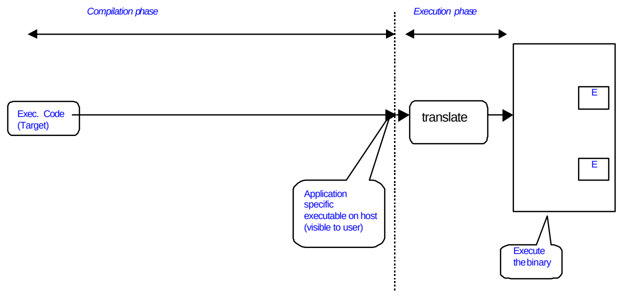

Figure 3. Dynamically Compiled Simulation

B. *Statically Compiled Simulation*

Figure 2 shows a statically compiled simulation system. In this technique, the simulator takes advantage of the any *a priori* knowledge of the target executable and performs some of the activities at compile time instead of execution time. Using this approach, a simulation compiler generates host code for instruction fetch, decode and operand reads at compile time. As an end product, it generates an application specific host binary in which only the execute phase of the target processor is unresolved at compile time. This binary is expected to execute faster, as repetitive actions have been taken care of at compile time.

While this approach addresses some of the issues with interpretive simulators, there are further limitations. First, the simulation compilers typically generate C code, which is then converted to object code using the standard *compile→assemble→link* path. Depending on the size of the generated C code, the file I/O needed to scan and parse the program could well reduce the benefits gained by taking the compiled simulation approach. The approach is also limited by the idiosyncrasies of the host compiler such as the number of labels allowed in a source file, size of switch statements etc. Some of these could be addressed by directly generating object code – however, the overhead of writing the application specific executable file to the disc and then re-reading it during the execution phase still exists. In addition, depending on the underlying host, the application-specific executable (which is visible to the user) may not be portable to another host due to different libraries, instruction sets, etc.

C. *Dynamically Compiled Simulation*

Figure 3 shows the dynamically compiled simulation approach. In this approach, target instructions are translated into equivalent host instructions (executable code) at the beginning of execution time. The host instructions are then executed at the end of the translation phase. This approach eliminates the overhead of repetitive target instruction fetch, decode and operand read in the interpretive simulation model. By directly generating host executable code, it eliminates the overhead of the compile, assemble, and link path and the associated file I/O that is present in the compiled simulation approach. This approach also ensures that the target executable file remains portable, as it is the only executable file visible to the user and the responsibility of converting it to host binary has been transferred to the simulator.

This paper is organized as follows. In Section II, we present a transparent multithreaded architecture that provides for scalable implementations. In Section III, we describe how our toolchain is generated. In Section IV, we provide simulation results. In Section V, we discuss related work. In Section VI, we draw conclusions.

II. SANDBLASTER PROCESSOR

An architectural function is *transparent* if its implementation does not produce any architecturally visible side effects. An example of a non-transparent function is the load delay slot made visible due to pipeline effects. Generally, it is desirable to have transparent implementations. Many architectures, however, are not transparent. When modeling an architecture, it is often desirable to model a specific implementation's performance. Because generating tools for a multiplicity of architectures and implementations is resource intensive, architecture description languages have been developed [2][3]. A characteristic of this approach has been the generation of both the tool chain and hardware description.

Sandbridge Technologies has developed the Sandblaster architecture for convergence devices [4][5]. Just as handsets are converging to multimedia multiprotocol systems, the Sandblaster architecture supports the datatypes necessary for convergence devices including RISC control, DSP, and Java code.

As shown in Figure 4, the Sandblaster multithreaded

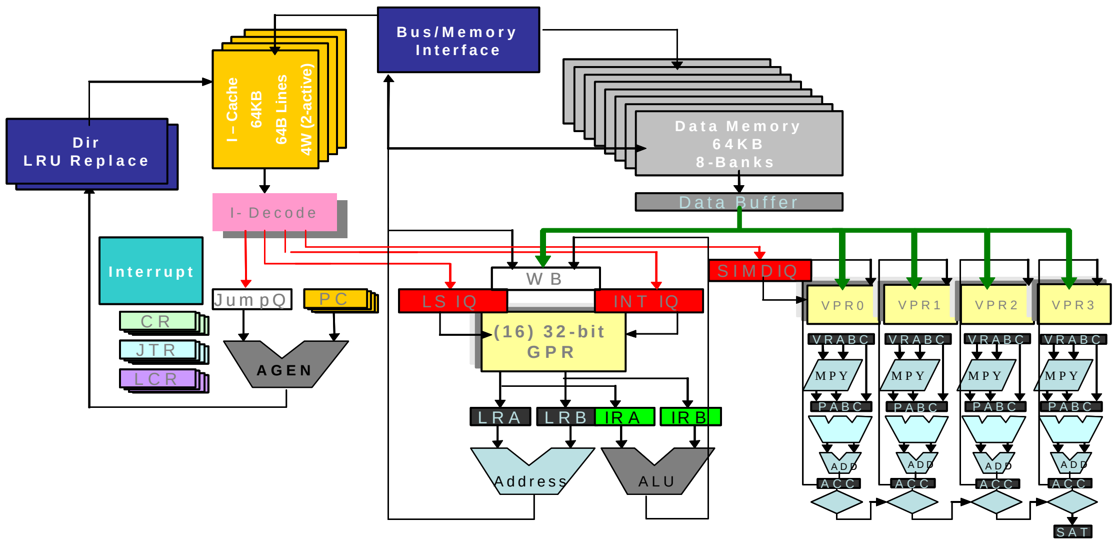

Figure 4. Sandblaster Multithreaded Processor

processor design includes a unique combination of modern techniques such as a SIMD Vector unit, a parallel reduction unit, and a RISC-based integer unit. Each processor core provides support for concurrent execution of up to eight threads. All state may be saved from each thread and no special software support is required for interrupt processing. The machine is partitioned into a RISC-based control unit that fetches instructions from a set-associative instruction cache. Instruction space is conserved through the use of compound instructions that may issue many operations..

The cache relieves the programmer of the need to move large programs into SRAM and avoids overlays that burden software systems. The cache also has the advantage that a programmer need only concern themselves with working set size (e.g. the dynamic code that predominantly executes), rather than the static instruction size that resides in flash or is downloaded dynamically over the air.

The data memory does not use a cache because in most broadband communications systems the data is streamed from A/D converters and passed on for further processing. Analogous to the instruction memory, the data memory also has eight independent banks for concurrent access by each thread. The complete memory system is unified to allow easy software access to any thread data.

Special care has been taken in the design of the memory subsystem to reduce power dissipation. The pipeline design in combination with the memory design ensures that all memories are single ported and yet the processor can simultaneously sustain nearly 4 taps per cycle (the theoretical maximum) in each hardware thread unit.

A RISC-based integer execution unit, depicted in the center of Figure 4, assists with control processing. Physical layer processing often consists of control structures with compute-intensive inner loops. A baseband processor must deal with both integer and fractional datatypes. For the control code, a register file with 16 32-bit entries per thread provides for very efficient control processing. Common integer datatypes are typically stored in the register file. This allows for branch bounds to be computed and addresses to be generated efficiently.

Intensive DSP physical layer processing is performed in the SIMD Vector unit depicted on the right side of Figure 4. Each cycle, four 16-bit vector elements may be loaded into the Vector File, while four pairs of 16-bit vector elements are multiplied and then reduced (e.g. summed), with saturation after each operation. The branch bound may also be computed and the instruction repeated until the entire vector is processed. Thus, retiring four elements of a saturating dot product and looping may be specified in as little as 64-bits, which compares very favorably to VLIW implementations.

An important power consideration is that the Vector File contains a single write port. This is in distinct contrast to VLIW implementations that must specify an independent write port for each operation in the VLIW instruction. Consequently, VLIW instructions, which are often up to 256-bits, may require register files with eight or more write ports. Since write ports contribute significantly to power dissipation, minimizing them is an important consideration in handset design.

A. Parallelism

To enable physical layer processing in software, the processor supports many levels of parallelism. Thread-level parallelism is supported by providing hardware for up to eight independent programs to be simultaneously active on a single Sandblaster core. This minimizes the latency in physical layer processing. Since many algorithms have stringent requirements on response time, multithreading is an integral technique in reducing latencies.

In addition to thread-level parallelism, the processor also supports data-level parallelism through the use of the SIMD Vector unit. In the inner kernel of signal processing or baseband routines, the computations appear as vector operations of moderate length. Filters, FFTs, convolutions, etc., all can be specified in this manner. Efficient, low-power support for data-level parallelism effectively accelerates inner loop signal processing.

To accelerate control code, the processor supports issuing multiple operations per cycle. Since control code often limits overall program speedup (e.g. Amdahl's Law), it is helpful to allow control code and vector code to be overlapped. This is provided through a compound instruction set. The Sandblaster core provides instruction level parallelism by allowing multiple operations to issue in parallel. Thus, a branch, an integer, and a vector operation may all issue simultaneously. In addition, many compound operations are specified within an instruction class such as load with update, and branch with compare.

Finally, the SB9600 product includes four Sandblaster processor cores per chip to provide enough computational capability to execute complete WCDMA baseband processing in software in real-time.

B. Java Execution

Future 3G wireless systems will make significant use of Java. A number of carriers are already providing Java-based services and may require all 3G systems to support Java [6]. Java, which is similar to C++, is designed for general-purpose object-oriented programming [7]. An appeal for the usage of such a language is its ``write once, run anywhere" philosophy [7]. This is accomplished by providing a Java Virtual Machine (JVM) interpreter and runtime support for each platform [8][9].

JVM translation designers have used both software and hardware methods to execute Java bytecode. The advantage of software execution is flexibility. The advantage of hardware execution is performance. The Delft-Java architecture, designed in 1996, introduced the concept of dynamic translation of Java code into a multithreaded RISC-based machine with Vector SIMD DSP operations [10][11]. The important property of Java bytecode that facilitated this translation is the statically determinable type state [7]. The Sandbridge approach is a unique combination of both hardware and software support for Java execution.

C. Interrupts

A challenge of visible pipeline machines (e.g. most DSPs and VLIW processors) is interrupt response latency. Visible memory pipeline effects in highly parallel inner loops (e.g. a load instruction followed by another load instruction) are not typically interruptible because the processor state cannot be restored. This requires programmers to break apart loops so that worst case timings and maximum system latencies are acceptable. This convolutes the source code and may even require source code changes between processor generations.

The Sandblaster core allows any instruction from any thread to be interrupted on any processor cycle. This is critical to real-time constraints imposed by physical layer processing. The processor also provides special hardware support for a specific thread unit to interrupt another thread unit with very low latency. This low-latency cross-thread interrupt capability enables fast response to time critical events.

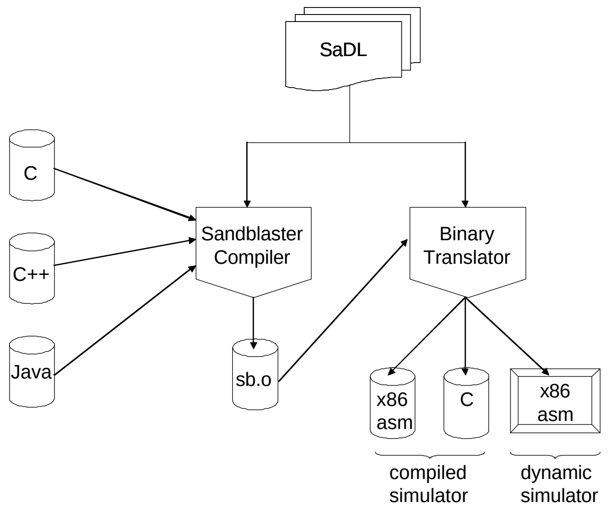

Figure 5. Tool Chain Generation

III. Tool Chain Generation

Figure 5 shows the Sandblaster tool chain generation. The platform is programmed in a high-level language such as C, C++, or Java. The program is then translated using an internally developed supercomputer class vectorizing, parallelizing compiler. The tools are driven by a parameterized resource model of the architecture that may be programmatically generated for a variety of implementations and organizations. The source input to the tools, called the Sandbridge architecture Description Language (SaDL), is a collection of python source files that guide the generation and optimization of the input program and simulator. The compiler is retargetable in the sense that it is able to handle multiple possible implementations specified in SaDL and produce an object file for each implementation. The platform also supports many standard libraries (e.g. libc, math, etc.) that may be referenced by the C program. The compiler generates an object file optimized for the Sandblaster architecture.

The tools are then capable of producing dynamic and static simulators. A binary translator/compiler is invoked on the host simulation platform. The inputs to the translator are the object file produced by the Sandblaster compiler and the SaDL description of the processor. From these inputs, it is possible to produce a statically compiled simulation file. If the host computer is an x86 platform, the translator may directly produce x86 optimized code. If the host computer is a non-x86 platform, the binary translator produces a C file that may subsequently be processed using a native compiler (e.g. gcc).

For the dynamically compiled simulator, the object file is

```
shr = opcode("shr", opcode = 0xa4, format = (Rt, Ra, Rb),
    resources = binop_resources(),
    jx86_exec = jx86_shu_body("shrl") + jx86_intop_wback(),
    doc_full = "Shift Right",
    doc_stmt = [
      EOp( EGP("rt"), "<-", EOp(EGP("ra"), ">>", EGP("rb")) )
      ],
    doc_long = "The target integer register, rt, is set to the value of the
     input register ra right shifted by the value of rb; 0s are shifted in
     to the high bits."
  )
```

Figure 6. Example SaDL Python Descriptions

translated into x86 assembly code during the start of the simulation. In single-threaded execution, the entire program is translated and executed, removing the requirement for fetch-decode-read operations for all instructions. Dynamically compiled multithreaded simulation is more complicated and described in Section III.B.

Dynamically compiled single threaded simulation translation is done at the beginning of the execution phase. Regions of target executable code are created. For each compound instruction in the region, equivalent host executable code is generated. Within each instruction, sophisticated analysis and optimizations are performed to reorder the host instructions to satisfy constraints. When changes of control are present, the code is modified to the proper address. The resulting translated code is then executed.

A. *SaDL Description*

The SaDL description is based on the philosophy of abstracting out the Sandblaster architecture and implementation specific details into a single database. The information stored in the architectural description file can be used by various parts of the tool chain. The goal is to keep a single copy of the information and replicate it automatically as and when needed.

The key part of the architectural description language is a set of python files which abstract the common information. These files keep information about each opcode on the Sandblaster processor. The description of an opcode contains a number of attributes – for instance the opcode name, the opcode number, the format, and the input registers. In addition, it contains the appropriate host code to be generated for the particular opcode. These description files are then processed by a generator to automatically produce the C code and documentation. The produced C code is used by our just-in-time simulator and other tools.

Figure 6 shows an example of an opcode entry in our architecture description language. It contains the opcode name, number, format and the input resources. It also has calls to the functions (jx86_exec statement) that are called to implement the operation on the host platform. In addition, it contains both the mathematical description (doc_stmt) and the English description (doc_long) to document the opcode

B. *Dynamically Compiled Multithreaded Simulation*

Dynamically compiled multithreaded simulation is more complex than the single-thread case because multiple program counters must also be accounted for. Rather than translating the entire object file as one monolithic block of code with embedded instruction transitions, and then executing it, in the multithreaded case we begin by translating each compound instruction on an instruction-by-instruction basis. A separate piece of code manages the multiple pc's, selects the thread to execute, and performs calls to the translated instruction(s) to perform the actual operation. The fetch cycle for each thread must be taken into account based on the scheduling policy defined in the SaDL implementation parameters. Properly speaking, the thread scheduling policy need not be considered for logical correctness; however, it facilitates program debugging. Although fetching with multiple program counters has an effect on simulation performance, compiled dynamic multithreaded simulation is still significantly faster than interpreted simulation.

When the simulator encounters a particular opcode during simulation, it calls the appropriate C function (generated from the processed architectural description files) for that opcode, makes a syntactic and semantic check and generates the host code to be executed on the host processor. By using this approach, all the modifications to the architecture description are limited to a small set of files. This minimizes errors and maximizes productivity.

IV. RESULTS

Figure 7 shows the simulation performance of the ETSI AMR speech encoder on out-of-the-box C code [13]. The simulation speed was measured by taking the wall clock time needed to simulate a certain number of cycles on a 1 Ghz Pentium. The results show that the simulation performance of a single threaded fully optimized and vectorized AMR encoder is 25 million instructions per second. The performance degrades to 15 MIPS for 8 threads and then very slightly for additional threads. The degradation is due to the overhead of simulating multiple instruction counters. Since there are 8 hardware threads simulated, there is only the overhead of scheduling contributing to additional degradation.

Previously, we have compared the simulation speed of our approach with that of other DSP simulators [14]. Our approach is up to four orders of magnitude faster than current DSP simulators. Comparatively, we can simulate in real-time on a simulation model of the processor while other

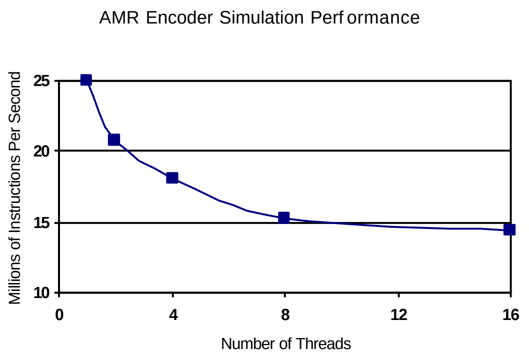

Figure 7. Simulation Results

approaches have difficulty achieving real-time performance on their own native platform. This provides a tremendous advantage in prototyping algorithms.

V. RELATED WORK

Automatic DSP simulation generation from a C++-based class library was discussed in [15]. Automatic generation of both compiled and interpretive simulators was discussed in [16]. Compiled simulation for programmable DSP architectures to increase simulation performance was introduced in [17]. This was extended to cycle accurate models of pipelined processors in [3]. A general purpose MIPS simulator was discussed in [18]. The ability to dynamically translate snippets of target code to host code at execution time was used in Shade [19]. However, unlike Shade, our approach generates code for the entire application, is targeted towards compound instruction set architectures, and is capable of maintaining bit exact semantics of DSP algorithms.

VI. CONCLUSIONS

We have presented a methodology for generating and simulating the Sandblaster architecture using the SaDL architecture description language. On a 1 Ghz Pentium processor, our dynamically compiled simulator is capable of simulating up to 100 million instructions per second for lightly optimized code. Fully optimized production DSP code simulates at roughly 25 million instructions per second and fully optimized multithreaded simulation provides nearly 15 million instructions per second.

Future optimizations include automatically generating MMX-style instructions. This should improve the performance of highly vector codes. In addition, for multithreaded simulation, simulation could proceed for all instructions until a synchronization point is reached and then fetch from the new PC. This would lower the overhead of fetching every cycle from a new PC.

REFERENCES

[1] G. Blaauw, and F. Brooks, Computer Architecture: Concepts and Evolution, Addison-Wesley, Reading, Massachusetts, 1997.

[2] A. Fauth, and A.Knoll, “Automated Generation of DSP Program Development Tools Using a Machine Description Formalism”,), IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-93) , Volume: 1 , pp. 457-460, Apr 27-30, 1993.

[3] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, “Lisa – Machine Description Language for Cycle Accurate Models of Programmable DSP Architectures”, Proceedings of the 36th Design Automation Conference, pp. 933-938, 1999.

[4] J. Glossner, T. Raja, E. Hokenek, and M. Moudgill, “A Multithreaded Processor Architecture for SDR”, The Proceedings of the Korean Institute of Communication Sciences, Vol. 19, No. 11, pp. 70-84, November, 2002.

[5] J. Glossner, E. Hokenek, and M. Moudgill, “Multithreaded Processor for Software Defined Radio”, Proceedings of the 2002 Software Defined Radio Technical Conference, Volume I, pp. 195-199, November 11-12, 2002, San Diego, California.

[6] J. Yoshida, “Java Chip Vendors Set for Cellular Skirmish”, EE Times, January 30, 2001.

[7] J. Gosling, “Java Intermediate Bytecodes”, ACM SIGNPLAN Workshop on Intermediate Representation (IR95), pp. 111-118, January, 1995.

[8] J. Gosling and H. McGilton, “The Java Language Environment: A White Paper”, Sun Microsystems Press, October, 1995.

[9] T. Lindholm and F. Yellin, “Inside the Java Virtual Machine”, Unix Review, Vol. 15, No. 1, pp. 31-39, January, 1997.

[10] J. Glossner and S. Vassiliadis, “The Delft-Java Engine: An Introduction”, Lecture Notes in Computer Science. Third International Euro-Par Conference (Euro-Par ’97), pp 776-770, Passau, Germany, August, 1997.

[11] J. Glossner and S. Vassiliadis, “Delft-Java Dynamic Translation”, Proceedings of the 25th EUROMICRO Conference (EUROMICRO ’99), Milan, Italy, September, 1999.

[12] K. Ebcioglu, E. Altman, and E. Hokenek, “A Java ILP Machine Based on Fast Dynamic Compilation”, IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel, January, 1997.

[13] European Telecommunications Standards Institute, Digital cellular telecommunications system, ANSI-C code for Adaptive Multi Rate speech Traffic Channel (AMR) (Ver 7.1.0), July, 1999.

[14] J. Glossner, D. Iancu, J. Lu, E. Hokenek, and M. Moudgill, “A Software Defined Communications Baseband Design”, IEEE Communications Magazine, Vol. 41, No. 1, pages 120-128, January, 2003.

[15] D. Parson, P. Beatty, J. Glossner, and B. Schlieder, “A Framework for Simulating Heterogeneous Virtual Processors”, Proceedings of the 32nd annual Simulation Conference, pages 58-67, April 11-15, 1999, San Diego, California.

[16] R. Leupers, J. Elste, and B. Landwehr, “Generation of Interpretive and Compiled Instruction Set Simulators”, Proceedings of the ASP-DAC '99, 18-21 Jan 1999, Wanchai , Hong Kong, Vol. 1, pp 339-342.

[17] V. Zivojnovic, S. Tjiamg, and H. Meyr, “Compiled Simulation of Programmable DSP Architectures”, Proceedings of the IEEE Workshop on VLSI Signal Processing, Sakai, Osaka, Oct 1995, pp. 187-196.

[18] J. Zhu and D. Gajski, “An Ultra-fast Instruction Set Simulator”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 10, Issue 3, June 2002, pp. 363- 373.

[19] R. Cmelik, “Shade: A Fast Instruction-Set Simulator for Execution Profiling”, Univ. Of Washington Technical Report # UWCSE 93-06-06.