An Integrated ARM and Multi-core DSP Simulator

Sharad Singhai Ming-Yung Ko Sanjay Jinturkar Mayan Moudgill John Glossner
Sandbridge Technologies Inc.
1 N Lexington Ave
White Plains, New York, 10601, USA
jglossner@sandbridgetech.com

ABSTRACT

In this paper we describe the design and implementation of a flexible, and extensible, just-in-time ARM simulator designed to run co-operatively with a multi-core DSP simulator on x86 hosts. The integrated simulator can boot ARM/Linux alongside another operating system running on DSP cores, thus truly supporting a heterogeneous multi-core operating environment. In addition, the simulator facilitates exploration of several system design parameters such as memory latencies, cache organization *etc.* via lightweight user-defined instrumentation.

We provide performance results and highlight the impact of design choices on our overall performance and design objectives. We also discuss implementation techniques and trade-offs between the competing requirements of simulation speed versus accuracy in a complex multi-core simulation environment.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimizations, Incremental compilers, Run-time environments*

General Terms

Design, Measurement, Performance

Keywords

ARM, just-in-time compilation, multi-core simulation, performance measurement, dynamic translation, Embedded architectures

1. INTRODUCTION

In this paper we describe the design and implementation of a flexible, and extensible, just-in-time ARM simulator designed to run co-operatively with a multi-core DSP simulator on x86 hosts. The integrated simulator can boot ARM/Linux alongside another operating system running on DSP cores, thus truly supporting a heterogeneous multi-core operating environment. In addition, the simulator facilitates exploration of several system design parameters such as memory latencies, cache organization *etc.* via lightweight user-defined instrumentation.

This multi-core simulator is actively being used in an industrial setting. It has been shipped to multiple wireless industry vendors, who are using it for physical layer simulation of GSM, WCDMA, TDS-CDMA, Wireless LAN, WiMax/WiBRO along with their protocol stacks. In this paper we focus on the design of ARM simulator and its performance.

The low-power multi-threaded Sandblaster processor [9] offers an efficient and flexible software-defined radio (SDR) platform for wireless communication applications, such as WCDMA with GSM/GPRS capability [6]. To streamline application development, a comprehensive bundle of software tools including simulator, compiler, assembler, debugger, and profiler is provided [5].

2. BACKGROUND

We developed a multi-core simulator for DSP and wireless applications. However, in this paper we focus on the ARM microprocessor. There are several existing ARM simulators. However, our motivation for developing an integrated multi-core simulator was to provide an ultrafast ARM and DSP co-simulation environment.

Our design goals and requirements were:

Instruction simulator A *functional* simulator as opposed to an architectural cycle-accurate simulator,

Speed A simulator that could execute millions of ARM and DSP instructions per second on an x86 host,

Integration with DSP simulator Seamlessly integrate and leverage existing DSP, memory, and peripheral simulation infrastructure,

Flexibility of simulation Easily support different ARM architecture versions,

Detailed instrumentation Support easy instrumentation of user programs to identify bottlenecks, and

Multiple platforms Support multiple host environments. We were interested in hosting the simulator on Linux as well as on Windows in a production environment.

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

Both our DSP and ARM simulators utilize *just-in-time (JIT)* translation technology, which refers to the dynamic translation of target instructions (ARM in our case) to the host instructions (x86 in our case). Just-in-time means that the target instructions are translated *just* as they are needed for execution. Thus JIT compilation typically interleaves instruction translation with the instruction execution and is incremental in nature.

JIT compilation[1] improves the runtime performance of a program, and has been testified to be practically effective. We also utilize JIT compilation to accomplish integration of two almost independent simulators for heterogeneous architectures, *i.e.*, ARM and Sandblaster DSP.

3. SIMULATOR DESIGN

In this section we describe the details of our simulator design and implementation. We discuss our techniques for fast ARM instruction decoding and tight x86 code generation.

3.1 Simulation Modes

Our integrated simulation can proceed in three *modes*, which differ in terms of increasing level of detail and associated complexity.

Mode 1: Straight-line ARM application code In this simulation mode, an ARM application executes in standalone mode, with only a subset of system calls being modeled. This is somewhat limited in its abilities, however, it works at the fastest speed with the MMU being off.

Mode2: ARM code with system-level simulation In this mode, the ARM code assumes existence of peripherals and Linux operating system, thus simulating the whole environment including exceptions and memory latencies. It supports the MMU but not DSP cores.

Mode 3: ARM code along with DSP In this most detailed (and the most complex) simulation mode, ARM programs executes concurrently with the multi-core DSP with two distinct operating systems. The DSP runs a proprietary operating system and ARM runs Linux operating system with MMU on.

These three simulation modes have implications for the simulator design. Figure 1 shows the JIT compiled instructions for the ARM only simulation modes versus mode 3, when an intervening DSP instruction (C1) must be executed between the two ARM instructions (A1 and A2). In the ARM only simulation (modes 1 and 2), the x86 instructions continue without having to save/restore x86 state for the next ARM instruction, thus resulting in greater speedup.

3.2 Just-in-time compilation

In this section, we describe the details of our JIT compilation for converting ARM instructions to x86 instructions.

We use a table-driven decoding method of ARM instructions. ARM 32-bit instructions have a regular pattern and the opcode can be determined by looking at only a small subset of 32 bits. Figure 2 shows the layout of the ARM 32-bit instructions. We needed to decode only shaded 12 bits in order to find the opcode.

[1]We use the terms JIT compilation and JIT translation interchangeably.

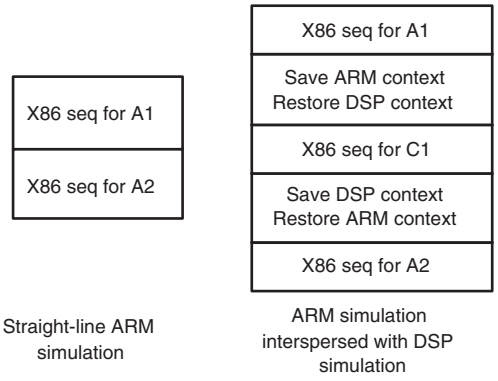

Figure 1: Mixed ARM and DSP JIT compilation

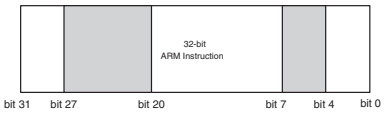

Figure 2: ARM 32-bit instruction decoding

We use Sandbridge-specific *system architectural description language* (sADL) to describe the opcodes. It is a higher level language which generates C header files for instruction decoding, code generation and disassembly. This sADL description is very succinct as proved by the fact that about 5000 lines of ARM v5TE DSP enhanced instruction set description generated more than 17000 lines of C code for JIT engine. In addition, various headers of about 10000 lines are also generated which includes an ARM disassembler for use in debugger.

Any regular ARM instruction can write to the program counter, complicating JIT engine. However, we can statically determine whether a particular instruction modifies pc or not—if it does, then we generate the appropriate sequence for a branch otherwise the pc is simply incremented.

3.3 Processor modes

ARM processors have seven different operating *modes*. These modes differ in what privileges are available in that mode as well as in accessible registers. There are 31 general-purpose 32-bit registers, however, in any given mode only 16 of them are visible.

We implement processor modes by maintaining separate register sets for each mode and updating the visible register set contents in response to a mode switch. Since the mode switch instructions are relatively rare, this approach incurs low-overhead. There are a few instructions which allow access to another mode's registers and we take care to ensure such accesses are done correctly.

3.4 Conditional execution

Almost all ARM instructions can be executed conditionally depending upon 4-bit condition flag, implying a total of 16 conditions. In fact, unconditional execution is just another condition. There are four (five in ARM architecture version v5 and above) condition flags in the current processor status register (CPSR). Each 32-bit ARM instruction has a 4-bit condition code field specifying under which combina-

tion of condition flags this instruction should be executed. In our experience, tight condition code generation turned out to be crucial for high performance.

3.5 Granularity of compilation

There are varying approaches to *when* a JIT compilation is performed in a JIT simulation. A completely demand driven approach is too expensive and an *a priori* approach is wasteful because many program paths may never be executed.

In our simulator we translate the ARM application code on a page-by-page basis, the page size in our case is 4 KB. At the end of each translated page we generate a call to the JIT compiler to translate the next page. Note that the control may still transfer out of translated page because of branches, function calls, exceptions, and so on.

An additional thorny issue is that of self-modifying code. If the corresponding x86 instructions are not immediately updated then the old instructions will be executed. Fortunately, such uses involve cache/tlb flush or invalidate operations for correctness. Our JIT compiler relies upon intercepting such cache flush/invalidate operations and we discard cached x86 code sequences. This guarantees that the JIT compiler is reinvoked.

3.6 System call implementation

For mode 1, we implement Linux operating system calls via traps from the generated target code to the C code. The ARM simulator makes a mode switch from the user mode to the system mode and forwards system calls to the underlying host operating system. This approach resulted in much faster and controlled system calls along with access to the host file system. For example, we can instrument system calls without modifying any of the original Linux kernel code. Note that the underlying operating system could be different than Linux, *e.g.*, Windows. We have an additional complexity related to simulating big-endian ARM/Linux on Windows with little-endian x86 host.

For modes 2 and 3, the system calls are not given any special treatment and are simulated as regular instructions. This mode faithfully simulates the operating system, but is correspondingly slower.

3.7 Peripheral models

In order to boot up ARM/Linux on simulator we needed to model at least a subset of peripherals found on our development hardware. Once the operating system is booted, the peripherals generate interrupts according to the simulated clock. We modeled the serial port (PL011), vectored interrupt controller (PL190), multi-port memory controller (PL172), general purpose input/output (PL061), real time clock (PL031), and flash memory conforming to the CFI (common flash interface).

3.8 Instrumentation

Our simulator also allows user-defined instrumentation to be linked in during the JIT compilation. Thus the instrumentation overhead remains very lightweight as a call is inserted only for the event of interest. Currently, a user defined program can intercept and measure dynamic instruction counts, cycles during external memory accesses, cycles during cache accesses, and static and dynamic opcode frequencies (as a post-pass). An important side-effect of this dynamically-linked instrumentation call infrastructure is that the amount of code the user has to write is typically very small. In this regard our simulator is similar to Hazelwood's simulator [7].

3.9 Simulating memory

We model discontiguous SDRAM (with holes in address space) which requires special handling. For a memory load (or store), if the base is always zero, we can directly load from the given offset. For discontiguous memory, we need to do more expensive bounds check and issue a data abort if a memory access lies outside of the discontiguous memory space.

We also support memory-mapped peripherals via special memories, which instead of loading from the simulated memory, use function pointers to load/store/initialize the corresponding range of memory addresses. Thus a new peripheral model can be easily added. We run ARM in big-endian, however, our host platform, Intel x86 platform, is little-endian. We need to do appropriate endian-conversion when loading code/data from the simulated memory.

We perform memory translation when the memory management unit (MMU) is enabled. We support this via address translation in software with the corresponding overhead for each access. However, a few simple applications not requiring MMU can run in Mode 1 at much faster speed.

3.10 Debugging support

Debugging programs in a JIT environment is a much harder task because the host instruction set (x86 in our case) may be completely different from the target instruction set (ARM in our case). Even the assembly level debugging is difficult because an application programmer might want to set breakpoint at a particular point in target code (ARM code) whereas only breakpoints available are for the host code (x86) on the host debugger. It should not be left to the user to decipher target state from the host state.

In order to facilitate low-level debugging, we implemented a debugger inside the simulator which has the ability to stop the ARM program at ARM instruction boundaries, synchronize the ARM state held in host x86 registers and display ARM register contents. We also implemented instruction and data breakpoints.

4. EXPERIMENTAL RESULTS

In this section, we demonstrate the ARM simulator performance results for real world applications.

We measure speed in terms of ARM instructions executed per second on a 2.4GHz Pentium 4 CPU. We believe that the raw instructions executed per second is a better measure than the cycles per second. For example, the cycle count is different in our simulation depending upon whether the other 4 DSP cores are accessing memory at the same time.[2]

In Figure 3 we display the performance of our simulator on a 2.4GHz Pentium 4 Linux machine simulating ARM processor in mode 1. We use commonly used wireless applications. We get average performance closer to 110 million instructions per second (MIPS). These benchmarks were compiled with ARM/Linux gcc with O3 level of optimizations.

[2]We model a single-port SDRAM shared among 4 DSP cores and ARM.

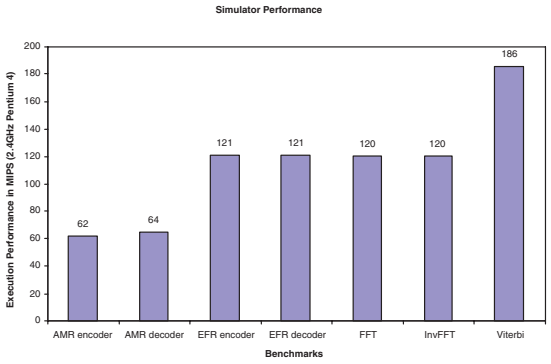

Figure 3: Simulator Performance

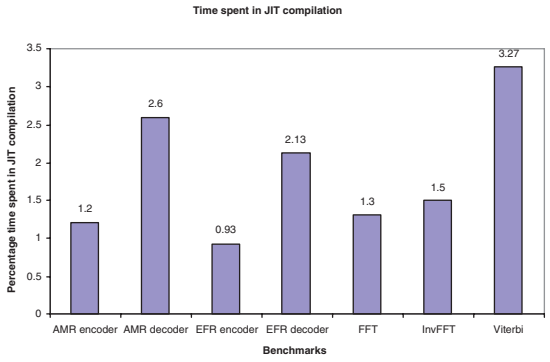

Figure 5: Percentage time spent in JIT compilation

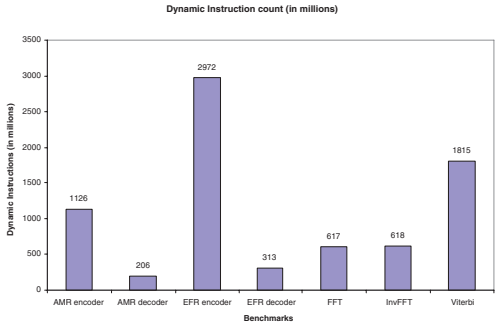

Figure 4: Benchmark Dynamic Instruction Counts

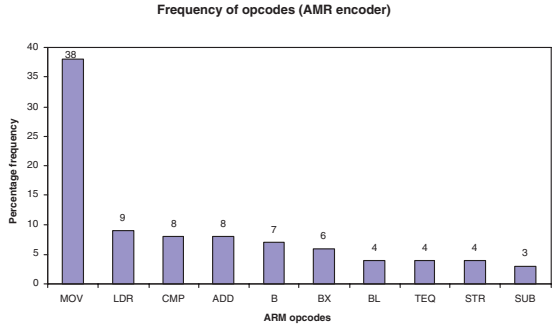

Figure 6: Frequency of opcodes in AMR encoder

Figure 4 describes the dynamic instruction count (in millions) for each benchmark application. The number of instruction executed varies from a low of about 200 millions to about 3 billion for these benchmarks. These dynamic instruction counts while certainly lower than the typical SPEC benchmarks, are nonetheless representatives of the communication algorithm kernels.

The benchmarks are summarized below.

AMR *Adaptive Multi-Rate speech traffic channels* is a speech coding algorithm which uses variable bit-rate encoding and decoding. The encoder processes an input file which contains 425 frames of binary speech data. The decoder uses this coded data as input and produces decoded speech output.

EFR *enhanced full rate speech traffic channels* is also a speech coding algorithm like AMR, but it uses a fixed rate for all input speech data. Coder uses input data, and outputs to a file and decoder uses the encoded data as input.

FFT It is *Fast Fourier Transform* commonly used in many algorithms for time domain to frequency domain conversion. Our particular version uses double-precision floating points and computes 512 point FFT. Inverse FFT (InvFFT) uses a similar computation, except that it computes the inverse Fourier transform.

Viterbi This benchmarks uses Viterbi algorithm for forward error correction as typically used in CDMA and GSM, 802.11 wireless LANs, and several other communication applications.

4.1 Time spent in JIT compilation

To measure the time spent in JIT compilation (as opposed to executing the code), we profiled the whole ARM simulator on host. Figure 5 shows the percentage time spent in compilation of the code versus executing the generated code. We observe that the process of JIT compilation itself is quite efficient as only about 3% of the total time is spent in JIT compilation in these benchmarks. The higher JIT compilation time of Viterbi is due to the higher percentage of control code.

4.2 Effect of branches on performance

To better understand the difference in performance between AMR and other benchmarks, we instrumented the AMR encoder and one representative benchmark, FFT to generate a dynamic count of opcodes executed. Figures 6 and 7 display the percentage frequency of top ten opcodes in AMR encoder and FFT respectively.

From these figures, we observe that the AMR encoder executes more frequent branch opcodes (`B`, `BX`, `BL`), for a total of 17%, while the FFT has only 6% branches (`B`, `BL`). These branches contribute to the slower speed of ARM instruction execution for AMR versus FFT. FFT has a regular loop-nest structure. We also observe that the load/store instructions themselves are not the bottleneck, because the FFT has 47% of load/store instructions (`LDR`, `STR`, `LDM`, `STM`), and

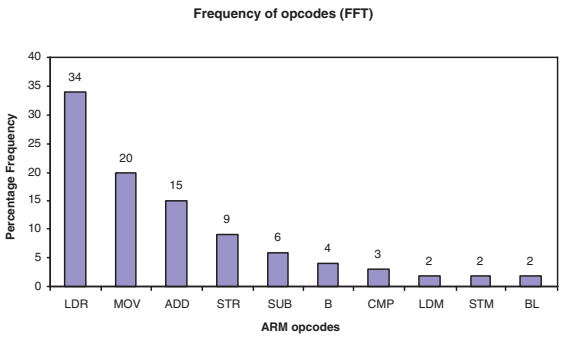

Figure 7: Frequency of opcodes in FFT

yet the number instructions executed per second is better than AMR.

5. RELATED WORK

The Sandblaster architecture has SIMD vector processing unit, and architecturally visible pipelines, similar to other DSP processors [4, 11].

Dynamic translation is an effective technique employed in compiler, simulation, and software instrumentation. Examples of dynamic compiler include Java JIT compiler [3], Shade [2], and ATOM [10]. A typical JIT example for instrumentation is the Pin project [8, 7], which allows users to insert function calls within a binary code to perform statistical analysis, performance profiling, and debugging.

System emulators provide extensible platforms when multiple systems are desired to run on host processors. For ARM architecture, the most widely deployed embedded processor, QEMU tool uses dynamic translation to emulate multiple ARM processors on several hosts [1].

6. FUTURE WORK

Our simulator provides essential elements for a whole system simulation, *e.g.,* boot Linux, and run multi-core simulation with ARM and DSP applications running concurrently. A simple cache invalidation approach is currently implemented for flushing code regions; a more sophisticated mechanism could be explored. Our ARM simulation environment provides a base software structure for simulating ARM architecture version up to ARM v5. We plan to extend to the other ARM processor versions.

Although our models are functionally correct, cycle accurate modeling of memory and peripheral devices is another potential direction. The task involves modeling of AMBA bus arbitration, multi-port memory controller, and various peripheral controllers.

Modeling of 16-bit Thumb or 8-bit Jazzelle instruction set is another direction to extend our ARM simulator. Because of our modular simulator design, the integration of vector floating point (*VFP*) co-processor model should be relatively easy.

7. SUMMARY

In this paper we describe the design and implementation of an ultrafast, flexible, just-in-time ARM simulator running on x86 host. This simulator was developed as a part of an integrated system-level multi-core DSP and ARM simulator. We discuss various design alternatives and demonstrate that reasonable simulator performance can be achieved by a combination of techniques. Our experience has been that the JIT compiler itself is a small part of the overall system design and one must carefully weigh various alternatives to achieve overall system performance without sacrificing modularity and portability.

We would like to thank Sean Dorward for his contributions. We also thank many anonymous reviewers for their extremely useful suggestions.

8. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference — FREENIX Track*, pages 41–46, 2005.

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[3] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. Compiling Java just in time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3):36–43, May/June 1997.

[4] J. Fridman and Z. Greenfield. The TigerSHARC DSP architecture. *IEEE Micro*, 20(1):66–76, 2000.

[5] J. Glossner, S. Dorward, S. Jinturkar, M. Moudgill, E. Hokenek, M. Schulte, and S. Vassiliadis. Sandbridge software tools. In *the 5th Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 269–278, Samos, Greece, July 2005.

[6] J. Glossner, D. Iancu, J. Lu, E. Hokenek, and M. Moudgill. A software defined communications baseband design. *IEEE Communications Magazine*, 41(1):120–128, January 2003.

[7] K. Hazelwood and A. Klauser. A dynamic binary instrumentation engine for the arm architecture. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pages 261–270, Seoul, Korea, October 2006.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[9] M. J. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis. A low-power multithreaded processor for software defined radio. *Journal of VLSI Signal Processing*, 43(2–3):143–159, June 2006.

[10] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. *SIGPLAN Notices*, 39(4):528–539, 2004.

[11] O. Wolf and J. Bier. StarCore launches first architecture — Lucent and Motorola disclose new VLIW-Based approach. *Microprocessor Report*, 12(14):1–4, October 1998.