Ultra Fast Cycle-Accurate Compiled Emulation of Inorder Pipelined Architectures*

Stefan Farfeleder[1], Andreas Krall[1], and Nigel Horspool[2]

[1] Institut für Computersprachen, TU Wien, Austria
{stefanf, andi}@complang.tuwien.ac.at
[2] Department of Computer Science, University of Victoria, Canada
nigelh@uvic.ca

Abstract. Emulation of one architecture on another is useful when the architecture is under design, when software must be ported to a new platform or is being developed for systems which are still under development, or for embedded systems that have insufficient resources to support the software development process. Emulation using an interpreter is typically slower than normal execution by up to 3 orders of magnitude. Our approach instead translates the program from the original architecture to another architecture while faithfully preserving its semantics at the lowest level. The emulation speeds are comparable to, and often faster than, programs running on the original architecture. Partial evaluation of architectural features is used to achieve such impressive performance, while permitting accurate statistics collection. Accuracy is at the level of the number of clock cycles spent executing each instruction (hence the description *cycle-accurate*).

1 Introduction

Emulation of instruction sets of different architectures is common. Originally, all emulators were interpreter-based. An interpreter mimics the execution of a standard computer by repeatedly fetching an instruction, decoding that instruction, and then executing it. The implementation is straightforward and allows insertion of monitoring code into the interpreter to gather any desired statistics. SimpleScalar and some other modern simulators still use interpretation because it allows cycle-accurate emulation of all features of today's complex architectures with out-of-order instruction execution [1].

The biggest disadvantage with interpreters is their extremely slow execution speed, which can be three to five orders of magnitude slower. Improving emulation speed is clearly desirable. In this paper, we describe techniques which achieve a speed-up by about three orders of magnitude — making the emulated program on a PC faster than on the original architecture.

2 Related Work

One technique for improving emulation speeds is memoization. Micro architecture states and the resulting simulator actions are cached. Then the emulation can be "fast

* This research was supported in part by Infineon and the Christian Doppler Forschungsgesellschaft.

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 222–231, 2005.

forwarded" whenever a cached state is reached. Schnarr and Larus [2] improved the speed of FastSim by 5 to 12 when emulating an architecture similar to a MIPS R10000. The speed can be further improved by using subroutine threaded interpreters which cache changed program parts [3].

Translating emulators are orders of magnitude faster than interpreters. Binary translation was first used for functional simulation of other architectures. A static binary translator takes a complete program, determines the program structure and translates the program into an equivalent one on the host architecture. Problems arise when indirect branches cannot be resolved at compile time or self-modifying code is used. A solution is to combine the translated program with an interpreter which is used in such a case. Binary translators have been successfully used for the simulation of the IBM 370 architecture [4] and for the migration of programs from the MIPS architecture to the Alpha architecture [5]. In contrast, dynamic binary translators convert short sequences of linear code into native code of the host architecture at runtime. This is the approach embodied in the Transmeta Crusoe architecture [6].

Shade [7] performs functional emulation and instrumentation, where collecting traces and similar information incurs a 2.8 - 6.1 slowdown. Embra [8] is a functional CPU model in SimOS and runs about 10 to 30 times slower by translating target instructions into the native instructions of the host. Bintrans [9] is a retargetable binary translator. From a description of the source and target architectures, a dynamic binary translator is automatically generated which executes programs between 1.8 and 2.5 times slower than the original.

Binary translation is tied to a fixed host architecture. Compiled emulation is more flexible because it generates C (or other high-level) source code for the emulated program. The compiler can optimize away most of the intermediate computations and thus improve performance. Mills et al. [10] generate one function for the complete program implementing branches by a switch statement. Amicel and Bodin [11] used assembly language source as the input language and generated C/C++ machine code. Retargetable compiled emulation has been successfully applied by Pees et al. [12].

3 The xDSPcore Processor Architecture

The simulated processor, xDSPcore [13], is a five-way variable-length very long instruction word (VLIW) load/store digital signal processor (DSP) with pipelined inorder execution. Up to five instructions are executed in each cycle. It supports some common extensions for the DSP domain, such as SIMD (single instruction multiple data) instructions, multiply-accumulate instructions, various addressing modes for loads and stores, fixed point arithmetic, predicated execution, etc. The processor's register file consists of two banks, one for data registers, the other for address registers. Each data register is 40 bits wide, but can also be used as a 32 bit register, or as two registers of 16 bit width ("shared registers", "overlapping registers", "register pairs").

The xDSPcore is a pipelined architecture. Some instructions need more than one execution stage. Register operands are read at the beginning and written at the end of the pipeline stage where they are needed. Branches have delay slots which can be filled with any instruction bundle. The xDSPcore's hardware loop instructions allow a fixed

number of repetitions of a piece of code without having to manage the loop counter in the code itself.

The simulated processor can make two memory accesses per cycle if they are to different banks, otherwise an additional memory access cycle is needed. There is no data cache, but there is an instruction buffer. The instruction buffer minimizes memory accesses and thus reduces power consumption on the xDSPcore. It has eight slots. Each slot holds one fetch bundle, which consists of four instruction words, plus an *executed bit*. The executed bit is set after all four of the instruction words are executed. The slot can be recycled and its contents overwritten by another instruction bundle only after the executed bit has been set. The xDSPcore's fetch unit reads one fetch bundle per cycle and writes it in a round-robin manner to the next slot in the buffer, omitting the write if that bundle is already cached or if the buffer slot does not have its executed bit set. A second unit, the aligner unit, reads four fetch bundles from the buffer and issues a stall if an instruction word needed for the next instruction bundle is missing.

4 Simulator Details

The requirements of our simulator were:

- fastest possible execution,
- cycle and state accurate,
- debugger support (single stepping, breakpoints),
- convenient architecture specification,
- portability (should run on common 32 and 64 bit computers).

The performance and portability requirements require compiled emulation. The assembly language source of the program to be emulated is translated into an equivalent C program which emulates the whole functionality of the simulated architecture. Despite difficulties caused when emulating a pipelined parallel architecture, basic blocks and loops are used as translation units. To handle unpredictable computed jumps and to support debugging, a full interpreter is integrated with the compiled emulator. Control is passed back and forth between the two components as required. The interpreter has a GUI which displays assembler source, and supports single-stepping and breakpoints.

For extending the architecture and for easy retargeting to other architectures, the syntax and semantics of the instruction set are specified in a XML configuration file. In the following sections, we describe how various problems in the emulator are solved.

4.1 XML Configuration File

Both the interpreter and the compiled emulator read their configurations from an XML file. It describes the complete instruction set and the hardware configuration for the register file, the pipeline, the instruction buffer, etc. The description of an instruction includes the execution semantics and additional text used for automatic documentation generation and to describe calling conventions. Figure 1 shows a slightly simplified and edited version of the XML description of the `ld` (Load) instruction. The instruction reads the value of an address register at the beginning of stage `EX1`, adds 2 to the register

```
<instruction>
  <mnemonic>ld</mnemonic>
  <operands>
    <operand>ADDR_REG</operand>
    <operand>LX_DX_RX_REG</operand>
  </operands>
  <syntax>(op1)+, op2</syntax>
  <semantics>
    <execute>READ_OP1</execute>
    <execute>MOD_OP1</execute>
    <execute>MEM_READ</execute>
    <execute>WRITE_OP2</execute>
  </semantics>
</instruction>
```

```
<map key="READ_OP1">
  <timing>EX1,begin</timing>
  <code>tmp1 = %op1</code>
  <code>tmp2 = %op1 + 2</code>
</map>
<map key="MOD_OP1">
  <timing>EX1,end</timing>
  <code>%op1 = tmp2</code>
</map>
<map key="MEM_READ">
  <timing>EX2,begin</timing>
  <code>tmp3 = mem[tmp1]</code>
</map>
<map key="WRITE_OP2">
  <timing>EX2,end</timing>
  <code>%op2 = tmp3</code>
</map>
```

Fig. 1. `ld` instruction with timings in the XML file

at the end of `EX1`, uses the old value as the address for a memory read at the beginning of stage `EX2` and stores the read value into another register at the end of the stage.

The identifiers within the `<execute>` elements reference other places in the XML file (shown in Figure 1), where the timings and the code that has to be generated for such an instruction part are stored. This separation of concerns facilitates maintenance – since many instructions share common parts, changes can be made at a single place.

The `<operands>` and `<syntax>` elements shown in Figure 1 are used for the assembler front-end. After an assembler line is split into simple tokens, checks are made as to whether the syntax and the types of the operands match the information found here.

4.2 Dividing the Instruction Bundles into Basic Blocks

The instruction bundles are traversed to find all basic block leaders. A leader is an instruction bundle that meets one or more of the following requirements:

1. it is a target of a branch instruction,
2. it starts the body of a hardware loop, or
3. it follows a branch instruction or the end of a hardware loop body.

For those branch instructions that have a branch delay, the instructions in the branch delay slots are appended to the branch instruction's basic block. If an additional branch is executed in a branch delay slot, only the first instruction of the target basic block is executed. In this case, a duplicate basic block which contains only the first instruction is generated. Each of these basic blocks is translated into a single C function in the generated output. This keeps the functions small, resulting in short compilation times and good optimization by the C compiler.

EX1	begin	`tmp1 = r0`
		`tmp2 = r0 + 2`
	end	`r0 = tmp2`
EX2	begin	`tmp3 = mem[tmp1]`
	end	`l0 = tmp3`

Fig. 2. Code for `ld (r0)+, l0`

4.3 Generating Code for Instructions

Consider an actual instruction with real operands, like `ld (r0)+, l0`. The placeholders for the operands that were shown in Figure 1 are simply filled with the actual operands. Figure 2 depicts the code generated for this instruction. The identifiers starting with `tmp` in the table are temporary variables used to cache register values or computed values. The C compiler should optimize unnecessary copies away. These temporaries also solve interdependencies between different pipeline stages of overlapping instructions in an elegant way.

Many arithmetic instructions can be implemented by a single C operator. Other instructions like multiply-accumulate, bit insertion or saturated computations do not have direct C counterparts. They are implemented by groups of operations or small inline functions which are read from the XML file.

4.4 Control Flow

Each generated C function returns the number of the next basic block to be be executed. This number is used as an index into an array of function pointers to locate the next basic block's function. The compiled simulator's main loop has the following simple structure:

```
int bbnr = <number of starting block>;
while ((bbnr = bbptr[bbnr]()) >= 0) ;
```

A software stack simulates the hardware stack for subroutine calls. At a call, the number of the basic block following the call instruction is pushed onto the stack, the called function number is returned and is thus executed next. A return instruction pops a function number from the stack and returns it.

4.5 Instructions Crossing Basic Block Boundaries

Consider the assembler code show in Figure 3. Because the `EX2` stage of the `ld` instruction is executed at the same time as `movr`'s `EX1` stage and because register `l0` is written at the end of a cycle, register `l1` receives `l0`'s old value. Therefore executing the whole `ld` instruction at the end of the basic block which contains the `br` instruction would give wrong results. To resolve these conflicts, the code fragments of `ld`'s `EX2` stage are moved into the basic block that begins with the label `foo:` and will be executed there in the correct order. The decision whether those moved code parts need to be executed is determined by a global variable that remembers the last executed basic block.

Basic blocks can be duplicated to improve performance. For every predecessor P_i of basic block B which has leftover pipeline stages, a specialized version B_i of basic block

```
 br foo
 nop
 ld (r0)+, l0
 ...
foo:
 movr l0, l1
```

Fig. 3. Overlapping between `ld` and `movr`

B is generated. It includes the code for the leftover pipeline stages. A global simulator switch determines the code generation scheme. In the previous example, the basic block is duplicated. Only one of them executes the second part of `ld`.

4.6 Simulating the Instruction Buffer

The addresses of the currently cached fetch bundles are stored in an array, as are the executed bits. At the beginning of each bundle, an attempt is made to insert the next fetch bundle's address into the array. A second table is used for a reverse-lookup because simulating the fully associative lookup would require up to eight comparisons per check. This second table associates each possible fetch bundle address with an index into the address array.

All instruction words between the program counter and the fetch counter are always held in the instruction buffer. Thus if one knows that the fetch counter is ahead of the instruction pointer by a sufficient amount, the check whether the instruction words needed for the execution of the next bundle are available can be omitted. To simulate this statically, the following strategy is applied. The program counter is initially set to the address of the first instruction bundle and the fetch counter is set to the address of the first fetch bundle. Program flow is simulated by adding four to the fetch counter and the amount of memory used by the instruction bundle to the program counter at every step. If the fetch counter does not exceed the program counter, there is no guarantee that the bundle is in the buffer. In this case, extra code is generated which performs a look-up for the needed address and to simulate a stall if it could not be found.

As already stated, executing a branch instruction sets the fetch counter and all executed bits. Code to simulate these actions is executed at the start of the destination basic block. When that destination block can be reached by both branching and by sequential execution, two versions of the block are compiled — one with and one without the extra code to set the fetch counter and the executed bits. Finally code to set the executed bit in the instruction buffer is inserted after all instruction words of a fetch bundle are executed.

Simulating the instruction buffer is expensive. Techniques to decrease the costs by computing extensive lookup tables at compile time are being explored.

4.7 Hardware Loops

The loop instruction is simulated by pushing a function pointer to the loop body's first basic block and the iteration count onto a stack. At the end of the loop, the counter is decremented; if it reaches zero, the following basic block gets executed, otherwise execution continues with the beginning of the loop body as found on the stack.

If a hardware loop consists of a single basic block, the simulator optimizes the loop into a C `for(;;)` statement, thus eliminating the overhead caused by a function call for each iteration and enabling the C compiler to apply further optimizations. If a hardware loop is sufficiently small to fit into the instruction buffer, a different optimization can be performed. The loop body is unrolled three times; the first copy simulates the buffer as described in the previous section for the first iteration, the second one repeats the body $n-2$ times. Since the instruction words are already buffered, the fetch simulation can be completely omitted. Finally the third copy of the body simulates the last iteration of the loop.

4.8 Memory Stalls

The xDSPcore has two memory ports, the X port covering the lower half of the data memory and the Y port covering the upper half. Two memory accesses are possible in a single cycle only if they do not use the same port, otherwise a pipeline stall occurs and the second access is deferred to the next cycle.

If two memory accesses are detected in a bundle, code to test whether the two memory addresses use the same port has to be inserted. If `tmp1` and `tmp2` are temporary variables holding the values of two address registers that are used to access memory, then the code to check if a stall occurs is similar to this:

```
if (!((tmp1 ^ tmp2) >> 15)) {
    ... /* issue a stall */
}
```

4.9 Collected Statistics

Each basic block has an associated counter which has to be incremented at runtime when entered. Using these counters, the dynamic number of executed instructions, bundles, the average number of instructions in a bundle, the frequency of each instruction, etc., can easily be computed. The number of memory stalls and aligner stalls are also counted. In addition, the emulator maintains extra counters for `.PROFILE` pseudo-instructions that are generated by the C compiler. They are used for feedback-driven optimization.

5 Experimental Results

Six sample programs, which represent typical applications for the xDSPcore processor, were used in our experiments: blowfish (symmetric block ciphering), dct8x8/dct32 (discrete cosinus transformations), g721 (voice compression), serpent (cryptographic algorithm) and viterbi (Viterbi decoder). The sizes of these programs and other characteristics are listed in Table 1. The dynamic parallelism column shows the average number of instructions executed in each cycle. The parallelism and the dynamic average basic block length have a significant effect on how efficiently the program can be emulated.

The left part of table 2 shows the speed of the six programs on a simple interpreter. Because statistics gathering has such a large effect on emulation speed, the speed is

Table 1. Characteristics of Test Programs

	Source size	Object size	Dynamic parallelism	Average basic block length
blowfish	25.8 kB	32 kB	1.91	14.38
dct8x8	43.9 kB	7 kB	1.85	7.48
dct32	35.8 kB	34 kB	2.14	8.73
g721	28.5 kB	5 kB	1.29	6.57
serpent	144.1 kB	46 kB	1.68	8.31
viterbi	36.6 kB	23 kB	1.21	216.85

Table 2. Emulation Speeds with an Interpreter and Compiler

	interpreted		compiled	
	with statistics	without statistics	with instr. buffer	without instr. buffer
blowfish	.083 MHz	.207 MHz	165 MHz	302 MHz
dct8x8	.082 MHz	.205 MHz	95 MHz	190 MHz
dct32	.071 MHz	.187 MHz	105 MHz	204 MHz
g721	.078 MHz	.198 MHz	78 MHz	259 MHz
serpent	.040 MHz	.208 MHz	120 MHz	258 MHz
viterbi	.094 MHz	.214 MHz	181 MHz	566 MHz

Table 3. Resources Needed to Create the Compiled Simulation

	Generation time (s)	Compile time (s)	C code size (kB)	Binary size (kB)
blowfish	3.22	3.06	316	257
dct8x8	3.32	4.51	421	396
dct32	3.27	5.13	780	542
g721	4.97	7.47	454	404
serpent	9.41	24.36	2081	1518
viterbi	3.50	60.85	475	411

shown with statistics gathering enabled and disabled. The right part of table 2 shows the execution speed of each of the programs when emulated with *Compiled Emulation*. The two columns show the cost of emulating the instruction buffer of the xDSPcore architecture. However it is necessary for guaranteeing cycle-accurate performance statistics. Statistics gathering has negligible effect on timings for the compiled emulation. Therefore, separate timing data is not shown for this case in the table.

The effective speed-up through using the compiled technique versus interpretation can be estimated by comparing the numbers in the “with statistics” column of Table 2 with the numbers in the “with instruction buffer” column of Table 2. The speed-ups range from 1000 to 3000. It can be seen that the largest speed-ups occur for the programs which have the longest basic blocks.

Finally, Table 3 shows the resources needed to generate and compile the emulated programs. Although the compiled programs are much larger than the original programs

on the xDSPcore platform, it should be remembered that they are executed on a much more powerful computer where memory is not a limitation. All measurements were made on an AMD Opteron 2Ghz CPU. The C code was translated by the Intel compiler with the `-O3` optimization level.

6 Conclusion

We have presented a novel approach for retargetable emulation of an architecture with some challenging features which include pipelining, a VLIW design, banked memory and an instruction cache. By generating C code which represents a translation of the original program at the basic block level, and which embodies the particular features of the emulated architecture, we have achieved impressive performance results. To our knowledge, we are the first to exploit partial evaluation of emulated features and extensive code duplication of the emulated program. The emulation speed is up to 3000 times faster than an interpreter while still maintaining a faithful simulation of the original architecture down to the number of clock cycles consumed.

References

1. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. Computer **35** (2002) 59–67
2. Schnarr, E., Larus, J.: Fast out-of-order processor simulation using memoization. In: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), ACM SIGPLAN, ACM (1998) 283–294
3. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: Proceedings of the 39th conference on Design automation, ACM Press (2002) 22–27
4. May, C.: Mimic: a fast system/370 simulator. In: Papers of the Symposium on Interpreters and interpretive techniques, ACM Press (1987) 1–13
5. Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., Robinson, S.G.: Binary translation. Communications of the ACM **36** (1993) 69–81
6. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO '03). (2003)
7. Cmelik, B., Keppel, D.: Shade: A fast instruction-set simulator for execution profiling. ACM SIGMETRICS Performance Evaluation Review **22** (1994) 128–137 Special Issue on Proceedings of the 1994 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94; 16–20 May 1994; Vanderbilt University, Nashville, TN, USA).
8. Witchel, E., Rosenblum, M.: Embra: Fast and flexible machine simulation. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. Volume 24,1 of ACM SIGMETRICS Performance Evaluation Review., New York, ACM Press (1996) 68–79
9. Probst, M.: Dynamic binary translation. In: UKUUG Linux Developer's Conference 2002. (2002)
10. Mills, C., Ahalt, S.C., Fowler, J.: Compiled instruction set simulation. Software – Practice and Experience **21** (1991) 877–889

11. Amicel, R., Bodin, F.: A new system for high-performance cycle-accurate compiled simulation. In: 5th International Workshop on Software and Compilers for Embedded Systems. (2001)
12. Pees, S., Hoffmann, A., Meyr, H.: Retargetable compiled simulation of embedded processors using a machine description language. ACM Transactions on Design Automation of Electronic Systems. **5** (2000) 815–834
13. Krall, A., Hirnschrott, U., Panis, C., Pryanishnikov, I.: xDSPcore: A Compiler-Based Configurable Digital Signal Processor. IEEE Micro **24** (2004) 67–78
14. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35** (2002) 50–58