Available online at www.sciencedirect.com
ScienceDirect

Journal of Systems Architecture 53 (2007) 501–510

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures ☆

Stefan Farfeleder [a], Andreas Krall [a,*], Nigel Horspool [b]

[a] Institut für Computersprachen, TU Wien, Argentinierstr. 8, Austria
[b] Department of Computer Science, University of Victoria, Canada

Received 29 March 2006; received in revised form 10 November 2006; accepted 10 November 2006
Available online 3 January 2007

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 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*).

Keywords: Instruction set emulator; Interpreter; Compiled emulation; Pipelined VLIW architecture

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 performing the actions implied by the instruction. The implementation is straightforward and allows insertion of monitoring code into the interpreter to gather any desired statistics. SimpleScalar and some other modern emulators still use interpretation because it allows cycle-accurate emulation of all features of today's complex architectures, even when they feature out-of-order instruction execution [1].

The biggest disadvantage with interpreters is their extremely slow execution speed, which is often three orders of magnitude slower. Even more extreme cases, up to five orders magnitude slower have been reported [2], where detailed performance

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

* Corresponding author. Tel.: +431 58801x18511; fax: +431 58801x18598.

E-mail addresses: stefanf@complang.tuwien.ac.at (S. Farfeleder), andi@complang.tuwien.ac.at (A. Krall), nigelh@cs.uvic.ca (N. Horspool).

1383-7621/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.11.003

characteristics are being measured and where the emulated system has been specified at a low level. Improving emulation speed is clearly a desirable goal. In this paper, we describe techniques which achieve a speed-up by about three orders of magnitude – often making the emulated program on a PC faster than on the original architecture.

In Section 2, we discuss related work. The architecture of the simulated processor is introduced in Section 3. In Section 4 we present our solution of a compiled emulation of a pipelined architecture where operations of an instruction are distributed over different pipeline stages. The evaluation of the emulator is contained in Section 5.

2. Related work

One technique for improving emulation speed is *memoization*. With this approach, micro architecture states and the resulting emulator actions are cached. Then the emulation can be "fast forwarded" whenever a cached state is reached. Schnarr and Larus [3] improved the speed of FastSim by a factor of 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 [4].

Translating emulators are orders of magnitude faster than interpreters. Binary translation was first used for functional simulation of other architectures [5]. A static binary translator takes a complete program, determines the program structure and translates the program into an equivalent one on the host architecture. However, 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 cases. Binary translators have been successfully used for the simulation of the IBM 370 architecture [5] and for the migration of programs from the MIPS architecture to the Alpha architecture [6]. In contrast, dynamic binary translators convert short sequences of linear code into native code of the host architecture at runtime. This is the approach used in the Transmeta Crusoe architecture [7].

Shade [8] performs functional emulation and instrumentation, where collecting traces and similar information incurs a slowdown by a factor of 2.8–6.1. Embra [9] is a functional CPU model in SimOS. It runs about 10 to 30 times slower, using a scheme where target instructions are translated into the native instructions of the host. Moreno et al. [10] implemented a two step approach for emulating VLIW architectures. Their Chameleon compiler translates the VLIW code to a virtual instruction set, ForestaPC. The ForestaPC instructions are subsequently translated to PowerPC instructions. The emulation overhead depends on the level of simulation; the authors report a slowdown by a factor of 7 to 10 if emulating the instruction set architecture, and an additional factor of up to 100 if the memory model and processor have to be simulated too. Bintrans [11] is a retargetable binary translator. The dynamic binary translator is automatically generated from a description of the source and target architectures. These translators generate programs for the target architecture. When the translated programs are executed, they require 1.8 to 2.5 times more CPU seconds than the original programs running on the source architecture.

Binary translation is tied to a fixed host architecture. In contrast, compiled emulation is a more flexible approach because it generates C (or some 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. [12] generate one function for the complete program, implementing branches by a switch statement. Amicel and Bodin [13] 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. [14]. The Sandbridge software toolset [15] has been used to emulate DSPs based on the Sandblaster processor. The toolset generates either assembly language or C code, and achieves emulation speeds of 15 to 100 million instructions per second on a Pentium with a 1 GHz processor. The emulation speed depends on whether the DSP code is multithreaded and how heavily it has been optimized.

Compiled emulation can be combined with hardware system simulation. Schnerr et al. [16] added FPGA boards for system simulation to a compiled instruction set emulator. With this extra help, they can emulate micro controllers in real time.

Compiled emulation bears some resemblance to the reverse compilation of assembly language programs to C. In [17], a control graph reconstruction algorithm for the TI C6x signal processor family was presented. This processor has a very long pipeline with explicit delay slots. Duplication of basic blocks is necessary to resolve these delay slots.

3. The xDSPcore processor architecture

The simulated processor, xDSPcore [18], 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 (see Fig. 1) consists of two banks, one for data registers, the other for address registers. Each data register (A0..A7) is 40 bits wide, but can also be used as a 32 bit register (L0..L7), or as two registers of 16 bit width (D0..D15) ("shared registers", "overlapping registers", "register pairs"). Address registers (R0..R7) are 16 to 32 bits wide.

The xDSPcore is a pipelined architecture. Load and multiply 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. Branch instructions have two delay slots which can be filled with any instruction bundle. The xDSPcore's hardware loop instructions allow a fixed number of repetitions of a fragment of code without having to manage the loop counter in the code itself.

The xDSPcore 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. The buffer 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 have been 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. A branch instruction sets the fetch counter that points to the next four instruction words that should be fetched to the branch target and sets the executed bits associated with all slots.

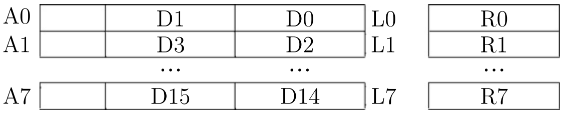

Fig. 1. Register File.

4. Simulator details

Emulation at the statement level is slow. We therefore treat basic blocks as single units for emulation purposes. As far as we are aware, we are the first to perform functional and accounting simulation for basic blocks and loops. Fallback to an interpreter, which we call *interpreted emulation*, is used to support breakpoints placed at single instructions, to support single stepping through the code, and to handle some indirect branch instructions.

The requirements of our emulator were:

- the fastest possible execution,
- cycle and state accurate emulation,
- debugger support (single stepping, breakpoints),
- convenient architecture specification, and
- portability (it 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, the *compiled emulator* and the *interpreted emulator*, as required. The interpreted emulator has a GUI which displays assembler source, and which 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 implementation 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. It is not only used for emulator generation, but also to generate the compiler, documentation and parts of the hardware. The description of an instruction includes the execution semantics and additional information used for automated translation to C, for documentation generation, and to describe calling conventions. Fig. 2 shows a slightly simplified and edited version of the XML description of the `ld` (Load) instruction. ⟨`timing`⟩ elements describe the pipeline behaviour of an instruction, ⟨`code`⟩ elements contain fragments of the semantics specification of an instruction. The instruction reads the value of an address register at the beginning of stage `EX1`, adds 2 to the register 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 Fig. 2), 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 Fig. 2 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 headers. A header is an instruction bundle which 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

```
<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. 2. `ld` instruction with timings in the XML file.

EX1	begin	`tmp1 = R0`
		`tmp2 = R0 + 2`
	end	`R0 = tmp2`
EX2	begin	`tmp3 = mem[tmp1]`
	end	`L0 = tmp3`

Fig. 3. Code for `ld (R0)+, L0`.

the functions small, resulting in short compilation times and good optimization by the C compiler.

4.3. Generating code for instructions

Consider an actual instruction with real operands, such as `ld (R0)+, L0`. The placeholders for the operands that were shown in Fig. 2 are simply filled with the actual operands. Fig. 3 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.

A unique number is appended to the temporary names to avoid clashes with other instructions. With long basic blocks this can create a lot of variables but they all have a very short live range, at most from the start of an instruction to its end.

The generator iterates over the phases (for the xDSPcore this is `begin` and `end`) of each cycle and copies the code pieces that are executed in that phase into the function. The code order within a phase does not matter. For example, the code of the instructions in bundle n with timing `EX2/begin` will be scheduled concurrently with those in bundle $n + 1$ and the timing `EX1/begin`. Immediately after that, the code for `EX2/end` of bundle n and `EX1/end` of bundle $n + 1$ will be generated.

Many arithmetic instructions can be implemented by a single C operator. Other instructions such as 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 executed. This number is used as an index into an array of function pointers to locate the next basic block's function. This enables a breakpoint to be set just by overwriting an entry in the array with a pointer to a special breakpoint function. The compiled emulator'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 shown in Fig. 4. 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. Temporary variables which hold information needed across basic block boundaries, and therefore across a function boundary in the generated C code, are replaced by global variables which resemble pipeline buffers.

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 B is generated. It includes

```
    br foo
    nop
    ld (R0)+, L0
    ...
foo:
    movr L0, L1
```

Fig. 4. Overlapping between `ld` and `movr`.

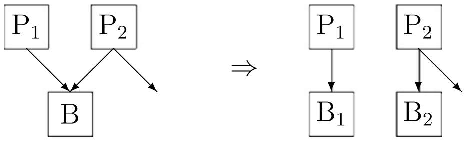

Fig. 5. Duplicating basic blocks.

the code for the leftover pipeline stages. A global emulator switch determines the code generation scheme. In the previous example (Fig. 4), the basic block is duplicated (see Fig. 5). Only one of them executes the second part of `ld`. The xDSPcore has few multicycle instructions. Therefore, code seldom needs to be duplicated.

4.6. Switching between the interpreted and compiled emulators

Some situations cannot be handled by a compiled emulator. A particular instance of such a situation occurs with a branch to an address computed at run-time. It may turn out that the target of the branch was not known to be a possible target when the source program was translated to C. Thus there would be no C function which can be called to continue emulation at the target instruction. The only practical solution requires subsequent emulation of the program by the interpreted emulator.

The implementation of an indirect branch proceeds as follows. First, the target address is computed. Then a hash table is searched to attempt to locate the basic block corresponding to that target address. If it is found, the compiled emulator retains control, using its main loop to invoke the C function corresponding to the basic block. If it is not found, the target address is not the beginning of a basic block or there are leftover pipeline stages which are not handled in the target basic block. (This latter case happens when the basic block containing the indirect branch was not recognized as a possible predecessor of the target basic block when the program was translated to C.) In all cases where the target address is not found in the hash table, control is transferred to the interpreter. The interpreter executes instructions one by one, returning control to the compiled emulator at the earliest opportunity, which would normally occur at the end of the basic block.

The biggest difficulty lies in transferring the complete pipeline state, which is buried in the compiled code, to the interpreter. Fortunately the temporary variables can be used to solve this issue. If execution is passed to the interpreter at the end of a basic block, all the information needed to fill its internal data structures for each execution unit's pipeline is represented by the values of the temporaries that are currently "live" and which are stored in the global variables which correspond to pipeline buffers. Taking the example from Fig. 3, if the switch to the interpreter is done after `ld`'s EX1 stage, the interpreted emulator should access the temporary variable `tmp1` because it will be needed for the memory access in the next cycle.

The other direction, where control is passed back to the compiled emulator, is handled similarly. Such a switch of emulation mode can only happen at a basic block boundary. The interpreter copies pipeline information into the global temporary variables which correspond to the pipeline buffers. Finally the hash table is searched for the number of the basic block which matches the program counter value. The compiled emulator is resumed in its main loop, where it calls the C function for the basic block.

Setting a breakpoint and single stepping through the program are both supported by the interpreted emulator. When a breakpoint is set inside a basic block B, the function pointer for B in the basic block array is replaced by a pointer to the breakpoint function. Thus, when the compiled emulator next attempts to execute that basic block, control is passed to the breakpoint function instead. The breakpoint function accesses a global table which records information about each basic block. The table includes the address of the start of each basic block and the basic block counters (see Section 4.10). The breakpoint function uses the information to restore the program counter and the pipeline state, it increments the basic block counter and continues execution in the interpreted emulator. At the end of the basic block, and assuming that single stepping is no longer being performed, control returns to the compiling emulator.

4.7. 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

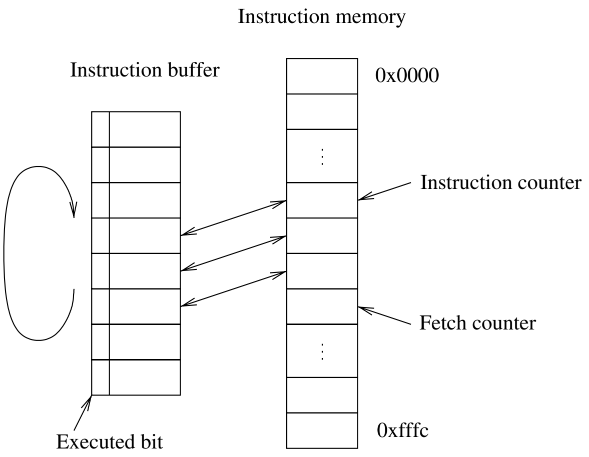

Fig. 6. Simulated Instruction Buffer.

bundle address with an index into the address array. Fig. 6 illustrates these data structures.

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 determine statically whether the check can be omitted, 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 in the instruction buffer and simulates 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.

4.8. 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 emulator optimizes the loop into a C `for(;;)` statement. Thus the overhead incurred by a function call on each iteration is eliminated, and the C compiler is able 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.

In a future version of the emulator, we plan to add simple loop analysis and to treat loops, which contain multiple basic blocks that fit into the instruction buffer, similarly to hardware loops.

4.9. 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 */
}
```

4.10. Collected statistics

Each basic block has an associated counter which has to be incremented at runtime when entered. The interpreted emulator has additional counters for instructions of partially executed basic blocks when the target of an indirect branch is inside a basic block. 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. They are: blowfish (symmetric block ciphering), dct8 × 8/dct32 (discrete cosine transformations), g721 (voice compression), serpent (cryptograhic algorithm) and viterbi (Viterbi decoder). The programs are originally in C and have been translated to assembly language by a highly optimizing compiler for the xDSP. The size of the assembly language source, the object code size, and other characteristics of these programs are listed in Table 1. The dynamic parallelism column shows the average number of instructions executed per cycle. The dynamic average basic block length gives the number of instructions per basic block weighted by the execution frequency. The dynamic parallelism and the dynamic average basic block length have both 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 our simple interpreting simulator. Because statistics gathering has such a large effect on emulation speed, the speed is shown with statistics gathering both enabled and disabled. The right part of Table 2 shows the execution speed of each of the programs when emulated with *Compiled Emulation*. A comparison between the two columns shows 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 to show its effect on emulation speed in the compiled case is not included in the table.

Table 2
Emulation speeds with an interpreter and compiler

	Interpreted emulation		Compiled emulation	
Statistics enabled?	Yes	No	Yes	Yes
Instr. buffer simulated?	Yes	Yes	Yes	No
blowfish	.083 MHz	.207 MHz	165 MHz	302 MHz
dct8 × 8	.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

The effective speed-up through using the compiled technique versus interpretation can be estimated from the data in Table 2. The numbers in the interpreted emulation column where statistics are enabled can be compared with those in the compiled emulation column where the instruction buffer is simulated. The speed-up factors range from 1000 to 3000. It can be seen that the largest speed-ups occur for the programs which have the longest basic blocks and low parallelism. (Viterbi has a parallelism of 1.21 and a very large average basic block size of 217.)

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

Table 1
Characteristics of test programs

	Source size (kB)	Object size (kB)	Dynamic parallelism	Average basic block length
blowfish	25.8	32	1.91	14.38
dct8 × 8	43.9	7	1.85	7.48
dct32	35.8	34	2.14	8.73
g721	28.5	5	1.29	6.57
serpent	144.1	46	1.68	8.31
viterbi	36.6	23	1.21	216.85

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
dct8 × 8	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

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] T. Austin, E. Larson, D. Ernst, SimpleScalar: an infrastructure for computer system modeling, IEEE Computer 35 (2) (2002) 59–67.

[2] J.A. Rowson, Hardware/software co-simulation, in: Proceedings of 31st ACM/IEEE Design Automation Conference, 1994, pp. 439–440.

[3] E. Schnarr, J. Larus, 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, pp. 283–294.

[4] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, A. Hoffmann, A universal technique for fast and flexible instruction-set architecture simulation, in: Proceedings of the 39th Conference on Design Automation, ACM Press, 2002, pp. 22–27.

[5] C. May, Mimic: a fast system/370 simulator, in: Papers of the Symposium on Interpreters and Interpretive Techniques, ACM Press, 1987, pp. 1–13.

[6] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, S.G. Robinson, Binary translation, Communications of the ACM 36 (2) (1993) 69–81.

[7] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, J. Mattson, 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, pp. 15–24.

[8] B. Cmelik, D. Keppel, Shade: a fast instruction-set simulator for execution profiling, ACM SIGMETRICS Performance Evaluation Review 22 (1) (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).

[9] E. Witchel, M. Rosenblum, Embra: fast and flexible machine simulation, in: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. ACM SIGMETRICS Performance Evaluation Review, vol. 24,1, ACM Press, New York, 1996, pp. 68–79.

[10] J.H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C.B. Hall, R. Miranda, S.-K. Chen, A. Polyak, Simulation/evaluation environment for a VLIW processor architecture, IBM Journal of Research and Development 41 (3) (1997) 287–302.

[11] M. Probst, Dynamic binary translation, in: UKUUG Linux Developer's Conference, 2002, pp. 1–12.

[12] C. Mills, S.C. Ahalt, J. Fowler, Compiled instruction set simulation, Software – Practice and Experience 21 (8) (1991) 877–889.

[13] R. Amicel, F.Bodin, A new system for high-performance cycle-accurate compiled simulation, in: 5th International Workshop on Software and Compilers for Embedded Systems, 2001, pp. 1–5.

[14] S. Pees, A. Hoffmann, H. Meyr, Retargetable compiled simulation of embedded processors using a machine description language, ACM Transactions on Design Automation of Electronic Systems 5 (4) (2000) 815–834.

[15] C.J. Glossner, S. Dorward, S. Jinturkar, M. Moudgill, E. Hokenek, M.J. Schulte, S. Vassiliadis, Sandbridge software tools, in: Embedded Computer Systems: Architectures, Modeling, and Simulation 5th International Workshop, SAMOS 2005, in: T.D. Hämäläinen, A.D. Pimentel, J. Takala, S. Vassiliadis (Eds.), Lecture Notes in Computer Science, vol. 3553, Springer, 2005, pp. 269–278.

[16] J. Schnerr, G. Haug, W. Rosenstiel, Instruction set emulation for rapid prototyping of SoCs, in: Proceedings of Design, Automation and Test in Europe (DATE '03), IEEE Computer Society, 2003, pp. 562–567.

[17] N. Bermudo, N. Horspool, A. Krall, Control flow graph reconstruction for reverse compilation of assembly langauge programs with delayed instructions, in: J. Krinke, G. Antoniol (Eds.), Fifth International Workshop on Source Code Analysis and Manipulation (SCAM'05), IEEE, Budapest, 2005, pp. 107–116.

[18] A. Krall, U. Hirnschrott, C. Panis, I. Pryanishnikov, xDSPcore: A Compiler-Based Configurable Digital Signal Processor, IEEE Micro 24 (4) (2004) 67–78.

Stefan Farfeleder is a member of technical staff at Catena. His research interests include compiler construction and digital signal processing. He received a M.Sc. degree from the Vienna University of Technology.

Andreas Krall is a professor of computer science at the Vienna University of Technology and leads the "Compilation Techniques for Embedded Processors" research laboratory. He received his Ph.D. and M.Sc. in computer science from the Vienna University of Technology. His research interests include compilers and interpreters with emphasis on code generation, compilation techniques for embedded processors, reverse compilers, binary translation, just-in-time compilers and cycle-accurate architecture simulation. He is a member of the ACM and the IEEE computer society.

Nigel Horspool is a professor of computer science at the University of Victoria. His primary research interests are in compiler construction and programming language implementation. He received a Ph.D. from the University of Toronto. He is a member of the ACM.