ISRN SICS-T--93/05-SE

Partial Translation

by

Peter Magnusson

SICS Technical Report
T93:05
ISSN 1100-3154

T93:05 ISSN 1100-3154

Partial Translation

Peter Magnusson
psm@sics.se

October 1993

Parallel Computer Systems
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA
SWEDEN

Abstract:

Traditional simulation of a target architecture by interpreting object code can be improved by translating the object code to an intermediate format. This approach is called interpretive translation. Despite a substantial performance improvement over traditional interpretation, a large part of the overhead is unnecessary. An alternative approach is block translation, where one or more simulated instructions are translated to directly executable code. This approach has several drawbacks.

We discuss the problems with block translation, analyse the overhead of interpretive translation, and describe a hybrid approach—*partial translation*—that combines the benefits of both approaches. Partial translation implements an intermediate format that supports the addition of run-time generated code whenever appropriate. The performance limit (slowdown) of interpetive translation is around 15, and real implementations have achieved 20-30. Partial translation will perform considerably better. Finally, we present results from an aggressive implementation of interpretive translation, and results from a proof-of-concept implementation of partial translation.

Keywords: Partial translation. Simulator. Interpreter.

Partial Translation

Peter Magnusson, SICS
October 1993

SICS Technical Report (T93:5)

Abstract

Traditional simulation of a target architecture by interpreting object code can be improved by translating the object code to an intermediate format. This approach is called interpretive translation. Despite a substantial performance improvement over traditional interpretation, a large part of the overhead is unnecessary. An alternative approach is block translation, where one or more simulated instructions are translated to directly executable code. This approach has several drawbacks.

We discuss the problems with block translation, analyse the overhead of interpretive translation, and describe a hybrid approach—*partial translation*—that combines the benefits of both approaches. Partial translation implements an intermediate format that supports the addition of run-time generated code whenever appropriate. The performance limit (slowdown) of interpetive translation is around 15, and real implementations have achieved 20-30. Partial translation will perform considerably better. Finally, we present results from an aggressive implementation of interpretive translation, and results from a proof-of-concept implementation of partial translation.

Introduction

Instruction-level simulators are a crucial component in developing and analysing computer architectures and system software. Instruction-level simulation allows a program written for a particular machine to be executed on a dissimilar machine. We call the former the *target* machine and the latter the *host* machine. Target programs execute in a simulation environment on the host machine and compute the same results as they would on the target.

Such simulators allow analysis of program behaviour in a manner impossible or impractical on real hardware. Traditional measurements are intrusive, either requiring insertion of special code in the program—thus interfering with the execution—or by implementing a limited form of sampling. Simulation allows performance statistics to be gathered unobtrusively. For computer architecture research, application studies can be done for conjectured hardware. Furthermore, instruction-level simulators are becoming an increasingly

Our focus

Our objective in simulator technology research is to find a common framework for both software and hardware work. We are aiming at a simulator core that is fast enough to run very large applications with realistic workloads. It should also be sufficiently accurate to support debugging of system software, and functional enough to support a variety of add-ons for gathering statistics and similar.

A discussion of previous work

Interpreting a source language, such as object code, is typically done in two phases: translating to an internal format and then interpreting this format. The traditional approach has been to bundle these phases into a single operation (Knuth 1973; Calingaert 1979). A more recent approach is to translate to an efficient internal format only once, and then interpret this format (Lang *et al.* 1986; Bedichek 1990). Adopting May's terminology, we can call the former *first-generation simulators* and the latter *second-generation simulators* (May 1987). First-generation simulators are mostly of historical interest.

The early second-generation simulators actually translated the target code to host code, and implemented an interface between the host code and the simulator. We shall call this approach *incremental translation.* A second alternative is *block translation,* e.g., we translate a program in parts or in its entirety. A more recent approach translates target instructions to an internal format, which is subsequently interpreted. This method, which we'll call *interpretive translation,* is a technique borrowed from language interpreters. The three alternatives are illustrated in figure 2.

Incremental translation

Translating individual instructions to native code brought the slowdown of simulators down to around 10, an improvement of one or two orders of magnitude over previous techniques.

The method is quite straight-forward (Lang *et al.* 1986). A fixed memory expansion is allocated for the code to be executed, allowing fast address mapping. This memory space is initialised with subroutine calls to a translation routine. When program flow reaches such a "cell," the target instruction is translated to one or more host instructions. If the expansion is more than the size of the cell, space is allocated off a heap.

For example, a register-to-register addition needs three registers. Which registers these are is known statically: code can be generated with fixed offsets into a register file structure. In contrast, an interpretive approach needs to determine which registers are used each time the instruction is executed.

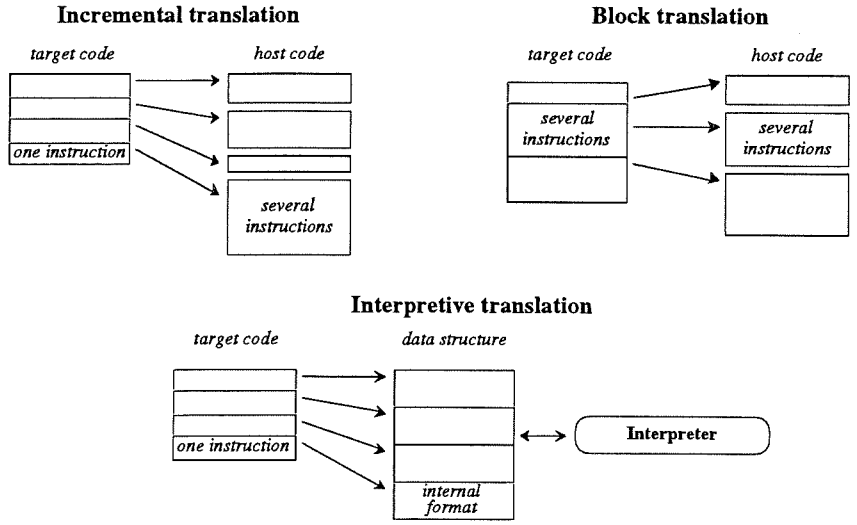

Figure 2 - Translation alternatives.

Block translation

The success of the early second-generation simulators prompted a "more is better" approach. The principal gain from block translation is that volatile state need not be stored back to main memory between each

However, interpretive translation has sharply reduced the cost of decoding and dispatching. Also, simpler processor designs has reduced the volatile state. The net effect is that interpretive translators have a slow-down of 20-30 (Bedichek 1990; Mills *et al.* 1991). Of this, the overhead is mostly due to saving and restoring volatile state. In addition to this reduced performance benefit, incremental and block translation have several drawbacks.

First, there is the memory expansion. The programs described have been small, and the author has found no references demonstrating incremental or block translation for simulation of a complete system (a modern operating system is easily several megabytes of code). This creates practical problems, as well as causing poor cache behaviour.

Second, neither scheme can easily handle dynamic performance factors such as a cache. Nor is it clear how to handle system-level simulation which requires, among other things, virtual memory, exceptions, dirty code, and asynchronous I/O. Also, simulating multiprocessors requires a fine grain of processor interleaving, down to a single cycle.

Third, with block translation, it is expensive to achieve very high performance. Optimisation of the translated code costs on the order of several thousand host instructions for each translated target instruction (May 1987).

Fourth, most incremental and block translation schemes are not portable, making it difficult to move a simulator to a new and faster platform.

Finally, the functionality of the simulator is restricted. It is not clear how to integrate functions like breakpoints, symbolic debugging, on-the-fly gathering of statistics, memory watch points, etc. without re-compiling the simulator every time.

Thus emblemished, block translation remains the most efficient approach. It is therefore a much desired goal to be able to combine the functionality of the interpretive translation with the performance of block translation.

The problems with interpretation

In an aggressive simulator based on interpretation, there are four steps to be done for each target instruction:

- simulate the instruction,
- update instruction pointers,
- check for an asynchronous event before the next instruction, and
- dispatch the next instruction.

The last three steps we'll refer to as the *epilogue*. In implementing a simulator on a SPARC, simulating a standard triadic instruction (such as register-register add) takes 8 instructions for the instruction itself, and 7 instructions for the epilogue. This yields a slowdown limit of this technique of 15.[1]

Of the 8 SPARC instructions used to simulate an 88100 instruction, only one is strictly necessary. The others are used for determining which registers are used, and saving/restoring them from the simulated register file. The 7 instructions in the epilogue only perform useful work if something interrupts instruction flow, which is generally not the case.

It is well known that these problems can be greatly reduced by block translation. The instruction pipeline can be handled at the time of translation. Instruction dispatch can be handled by falling through in the simulator code, i.e., by using the hardware of the host machine directly. Handling events is a problem. A common solution is checking events only rarely, such as once per translated block. This has several drawbacks, which we do not have space to describe in detail here. These include recreating errors, setting time breakpoints, and single stepping.

It would therefore seem that the interpretive solution is at odds with block translation. However, it is possible to obtain the best of both approaches in a single design. We call this approach *partial translation.*

---

[1]For purposes of discussion, we define slowdown limit as the proportion of host instructions per target instruction.

The example code is written in pseudo-assembler, since the discussion is processor independent. The example code above might occur inside a block copy routine. Each iteration would take 120-180 instructions in a simulator based on interpretive translation, i.e. assuming a slowdown of 20-30.

In interpreting this code, a large amount of the work done is unnecessary. We wish to eliminate as much as possible unnecessary work, namely:

- Instruction flow. Determining the next instruction implies a dispatch cost that can be eliminated by using the "normal" flow of the host processor: translated code will lie consecutively on the host machine.
- Event handling. All asynchronous events, and several other functions, are mapped to a single counter. The semantics of this counter is that it is decremented on each executed instruction, and an event handler called when it reaches zero. Since we know the length of the block (in the example, 5 instructions) we can decrement the event counter in blocks of 5. This will preserve the semantics as long as nothing within the translated block affects the event queue.
- Unnecessary write backs. Since each iteration overwrites the result of r5, r7, and r9, we can save effort by only writing back the values when we exit the loop.
- Reading/writing registers. Since the block will execute all-or-nothing (or only the first instruction), we can temporarily cache registers r5, r7, r9, and r13 in host registers.
- Branch delay slot. Within the loop, we know statically what the execution pipeline will be, and need generate code for branch delay slots only when necessary.

This covers most overhead involved in incremental translation. We listed them here in order to later show that they can all be handled in entry/exit code in the translated block.

We wish to generate a minimal amount of code, so only the common cases should be handled directly in the translated block. If the dynamics are complex, the compiled code needs to be exited in an orderly manner: registers, flags, and instruction pointers need to be updated to reflect the point of the code where the exit occurred.

It would be possible to implement all functionality in translated code. This would be detrimental to performance, however. The more code is generated for handling special cases, the less the cost involved in communicating back to a core simulator code. For instance, consider a TLB (translation look-aside buffer) miss. This occurs when a memory translation operation in the target architecture misses in the cache that contains the most recent translations (the TLB). Detecting a (possible) TLB miss requires little code. The advantage of generating code is to eliminate the "process switch" cost, i.e., moving between the generated code context and the (pre-compiled) simulator context. This cost becomes less of an issue the more complex the operation is. Furthermore, the simulator context has the advantage of being implemented in C and compiled with an optimising compiler. There is also the issue of host instruction cache performance. Large, generated blocks of code will lead to poor instruction cache performance on the machine running the simulator.

There are two ways of combining the threaded code model with block translation. Either the internal format includes a direct pointer to the compiled block, or we introduce a new instruction, TRANSLATED. This instruction takes as a parameter a pointer to the translated block, and handles generic entry/exit issues. In the former approach, the previous instruction dispatches the decoded block directly (by jumping to it), and entry/exit code needs to be compiled into every block. Both approaches have the advantage that re-entry needs no special checks.

The responsibility for maintaining consistency with the interpretive model can be split between the interpreter and the code block in several ways. For instance, accessing memory involves a virtual-to-physical translation on every access. An efficient simulation of virtual memory will have an optimistic path, and this would be compiled into the translated block. The responsibility of the translated code is to correctly handle one of three cases:

- handle the first instruction correctly, and dispatch for interpretation the next instruction,
- handle one or (if it is a loop) several iterations of the block, and
- fail, in which case it de-allocates itself, invalidates the translation of the first instruction, and re-dispatches the first instruction for interpretation.

```
                rSTATUS = <CONST>          ; set status bits correctly
                br <exit_0>                ; exit to service routine
                                           ;  (fall through)
exit_0:         rOP = upper<CONST>         ; load rOP with description of first
                rOP |= lower<CONST>        ;  instruction ("r7 = [r6]")
                br <sim_read>              ; and branch to service routine
exit_1:         rSTATUS = <CONST>          ; couldn't handle read, set status
                br <exit_0>                ; exit to service routine directly
exit_2:         rSTATUS = <CONST>          ; couldn't handle write, set status
                rEVENT = rEVENT - 2        ; read took 2 cycles (we assume)
                rPC = rPC + 4              ; point to next instruction
                br <exit> ; clean up
exit:           rCPU[20] = r1              ; restore registers
                rCPU[28] = r2
                rCPU[36] = r3
                rCPU[52] = r4
                rOP = [rPC]                ; get next instruction specification
                r1 = rOP & rMASK           ; get the service routine offset
                cmp rEVENT,0               ; check if an event has occurred
                br leq,<event>             ; deal with event
                br rCODE[r1]               ; otherwise dispatch next insruct.
exit_d:         call <dealloc_trans>       ; routine needs no params (uses rPC)
                [rPC] = 0; ; invalidate self
                rOP = 0;
                br rCODE                   ; force new translation
```

Some comments on the code.

<CONST> is some constant known at compile time. Some of the constants may not fit into immediates, so the instructions that use them may require one or more extra instructions to create the constant.

The translated block is meant to handle most cases, but not all. In this example, there are 5 exit circumstances, which for purposes of discussion we will call normal exit, exit 0, exit 1, exit 2, and exit and deallocate. They are described in the text below as they occur.

During translation of the block, we estimate that the block will take 8 cycles if executed (in this example, one cycle per instruction except memory accesses which take two). We first check if there are 8 or more cycles left, otherwise we abort to **exit 0**. Exit 0 dispatches the first instruction of the block as if it had been an interpreted instruction. It does this by loading the corresponding rOP value and branching to the interpretation code. Of course, it knows exactly where to branch to. The overhead of unsuccessfully attempting to dispatch the block is 5 instructions, of which two are branches and none are memory accesses.

The next case that the code checks for is if the CPU is in a legitimate state. The status bits in rSTATUS includes state such as carry, which are compared against a stored value. If the CPU state has changed, the generated code is no longer valid. The deallocating exit (exit_d) calls a routine that deallocates the space used by the code. This routine uses rPC to determine which space to remove. It does not, however, overwrite the block, and therefore the block can still execute! When the routine returns, it invalidates the translation in the intermediate code, and dispatches opcode "0", which corresponds to *not_decoded.*

The rSTATUS code may not be necessary. For instance, flags could be kept in a specific register—rFLAGS—and used (and updated) directly by the generated code.

The third check done upon entry is to assert that the instruction pipeline looks the same as the translation routine thought it would. The code generated in the translated block assumes a particular content in the pipeline. This is not likely to change, but this check is necessary for sake of correctness. It could be optimised away by dispatching this block of code directly only by other translated blocks, and checking the instruction pipeline only in the TRANSLATED pseudo-instruction. If this check fails, then the instructions are interpreted instead, so we branch to exit 0.

We don't want code expansion to get out of hand, so one approach might be to compile basic blocks. Frequent branch targets is a possible heuristic that is cheap to implement and would catch critical loops. That is, any basic block that is frequently branched to is translated.

An interesting effect of the block semantics is that they may overlap or include one another. This of course compounds the space dilemma, but could be useful for very tight code segments.

If the entry/exit code of blocks is cheap, it may be worthwhile to translate any sequence of instructions that do not change control flow or access memory.

Why does it work?

There are several characteristics of typical programs that benefit partial translation.

Executed code tends to have a very high degree in locality: the same registers tend to be used for the same purposes, a few instructions are very common, similar instruction sequences are generated, constants tend to be a low exponent of 2, branches are mostly on the same instruction page, memory accesses by a particular instruction tends to hit the same page each time, and a small amount of code do most of the work.

Prototype implementation

We implemented a prototype to study the practicality of this technique, and to eventually measure it's performance on large programs with realistic workloads. The prototype is initially aimed at running the Dhrystone (v2) benchmark program. Though too simplistic for performance measurements, it exercises a sufficient number of processor and memory functions to be a suitable test case.

The Dhrystone program is compiled with an 88k port of GCC, and linked with a trivial library for supporting simple Unix commands like printf()-since an operating system is not running under it. We modified the program slightly, locking it to 10000 iterations and skipping the timing code (which uses a timer and floating point instructions). These modifications affect only entry/exit code to the actual bench marking code, so it will not affect performance but served to simplify our first prototype.

The modified Dhrystone program uses only a handful of M88100 instructions: addu, cmp, ext, lda, mak, mul, or, subu, bb1, bcnd, divu, br, bsr, jmp, jsr, and 10 variants of store and load. Three instructions are only executed a handful of times: and, clr, and ff0. The first 10000 iterations of the benchmark executes 14971162 instructions. A dynamic count of these instructions are even more concentrated: 5 instructions and 4 variants of load/store represent 85% of instructions executed.

The prototype can interpret all these instructions. A subset can be translated to SPARC code, and correct entry/exit code generated for interfacing this code with the interpreter core. To illustrate, consider the following M88100 code (this code is from the library and sets up stack and calls main()):

```
5fe04020    or.u    r31,r0,16416
67ff0020    subu    r31,r31,32
5dc00000    or.u    r14,r0,0
248e4bbc     st     r4,r14,19388
5f400000    or.u    r26,r0,0
5dc00001    or.u    r14,r0,1
f77a600e    addu    r27,r26,r14
f79b600e    addu    r28,r27,r14
f7bc600e    addu    r29,r28,r14
c800019d    bsr     _main
f000d080    tb0     0,r0,128
```

The "bsr" command (branch to subroutine) is presumed to imply a major change in instruction flow, so it is not translated. The 9 instructions prior to the bsr instruction are translated to a single block of SPARC code. The code generated by the prototype appears below:

```
(a)     0x3a800  <end+93496>:     sethi   %hi(0x4000), %l5
(b)     0x3a804  <end+93500>:     sethi    %hi(0x40200000),%l7
(c)     0x3a808  <end+93504>:     sub  %l7, 0x20, %l7
(d)     0x3a80c  <end+93508>:     sethi   %hi(0), %l6
```

In (a) we create the only large constant used by the code. The M88100 processor can specify 16-bit immediates, whereas SPARC can handle only 12 bit unsigned. This constant is used by the simulation of "st". Next we simulate the first 88k instruction in (b), mapping r31 to SPARC register %l5. (c) and (d) simulate the "subu" and "or.u" instructions that follow.

Our first complex instruction is the store, the generated code for which begins at (e) and requires 8 instructions in this case. The generated code follows a different path for handling memory than described previously in the paper. Since blocks are translated on demand, the processor is in a correct state upon entry. If we "shadow" the instruction execution, we will know the register contents for the *first* execution of the memory access. We simply do a TLB lookup at translation time, and generate code to confirm that it is still this page being used. This is the *static* approach. A dynamic approach would keep some register translations in memory, or do a simple hash table lookup, for each memory access. Preferably, the code generation will adapt to how the code is used. We have yet to combine static with dynamic translation prediction. The first 6 instructions starting at (e) perform a comparison of the top 20 bits with the statically guessed logical page number, and aborts the code if this fails. The cleanup code generated upon exit, starting at (f), first store back the correctly executed code (namely, the changes to r31 and r14), and then dispatch the memory store handler. We will explain this exit code shortly.

If the store goes well, the next five instructions starting at (g) simulate five 88k instructions: two "or.u" followed by two "addu".

The code beginning at (h) is the normal exit handler. It first figures out which instruction to dispatch next (the absolute address of which it knows at the time of translation). It stores in %g1 the return address. It also reads the parameters of the next instruction into %g3, which is rOP. The updates to %g4 and %g5 correspond to setting rXIP and rNIP correctly.

Now we write back modified registers, starting at (i). This code block is unusual in that it modifies several registers. Finally, in (j), we dispatch the next instruction directly. This may or may not be another translated block.

The SPARC registers we overwrite in the generated code must not interfere with the interpreter. We make use of an extension to GNU C that allows auto variables to be allocated to specific registers. We expect the data-flow analysis of the compiler to save any other variables held in registers to memory when encountering a "goto (void *)rXIP;" statement in the interpreter.[2] Variables shared between interpreter code and generated code are stored in the global ("g") registers that are preserved across procedure calls. These are mostly reserved by the SPARC ABI, but while we execute code in the interpreter there won't be a collision.

The only information that needs to be communicated to the generated block are contained in these registers. These are:

- rEVENT - event counter. Contains the number of cycles to the next event.
- rOP - parameters. By default, these are initialised by the "caller". This is not used by the block currently, but needs to be updated on dispatching the next instruction.
- rCPU - current processor. Points to the data structure containing all processor specific information. The first part of the structure contains the user register file.
- rNIP and rXIP - the two stages of the pipeline (rFIP is implicit). rXIP points to the next instruction to be executed or the current one, and rNIP points to the prefetched instruction. By always updating rNIP on exit from a block or interpreter routine allows efficient and simple handling of branch delay slots.

Some basic functions are yet to be implemented. Primarily, the generated code does not check rEVENT or rNIP, though such code is easy to add.

The translation takes approximately 300 SPARC cycles (on a 40MHz IPX) per target instruction. The above block is thus output in 7.5 microseconds. The block executes in less than 40 SPARC cycles, including dispatching the next instruction or block. This implies a peak performance of around 9 MIPS of target instructions. A cursory static analysis of Dhrystone would indicate that 80% of instructions are in blocks of

[2]This is the mechanism by which GNU C jumps to a specific address. Addresses of labels are extracted with the "&&" operator. The resulting "(void *)" pointer can subsequently be handled as a normal pointer. The "goto" statement can jump to any void pointer.

Knuth, D. E. 1973. The Art Of Computer Programming Vol. 1, Fundamental Algorithms. Addison-Wesley, Reading, Mass.

Lang, T. G.; J. T. O'Quine; and R. O. Simpson. 1986. "Threaded Code Interpreter for Object Code." IBM Technical Disclosure Bulletin 28, no. 10 (March): 4238-4241.

Lewis, D. M. 1991. "A Hierarchical Compiled Code Event-Driven Logic Simulator." IEEE Transactions on Computer-Aided Design 10, no. 6 (June): 726-737.

Magnusson, P. 1993. "Efficient Simulation of Parallel Hardware." Proceedings of the International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS'93, SCS Simulation Series, Vol. 25, No 1, 1993..

May, C. 1987. "Mimic: a Fast System/370 Simulator." In Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques (St. Paul, Minnesota). 1-13.

Mills, C.; S. C. Ahalt; and J. Fowler. 1991. "Compiled Instruction Set Simulation." Software - Practice and Experience 21, no. 8 (Aug.): 887- 889.

Nash, H. 1989. "The Design and Development of a Software Emulator." In Digest of Papers, COMPCON Spring 1989. 34th IEEE Computer Society International Conference: Intellectual Leverage (IEEE Catalogue No. 89CH2686-41), 288-91.

Nussbaum , D. and A. Agarwal. 1991. "of Parallel Machines." Communications of the ACM 34, no. 3 (Mar.): 57-61.

Robertson, A. R. and R. N. Ibbett. 1991. "Simulation of the MC88000 Microprocessor System on a Transputer Network." In Distributed Memory Computing, 2nd European Conference, EDMCC2 Proceedings. (Munich, Gernmany, 22-24 April). Berlin, Germany, Springer-Verlag, 264-273.

Stallman, R. M. 1992. Using and Porting GNU CC, version 2.0 (15 February 1992), Free Software Foundation, Mass., USA.

Weicker, R. P. 1984. "Dhrystone: A Synthetic Systems Progrmming Benchmark." Communications of the ACM 27, no. 10, (Oct.): 1013- 1030.

Weicker, R. P. Dhrystone benchmark, C, version 2.0, Siemens AG, Postfach 3240, 8520 Erlangen, Germany.

Wirth, N. 1986. "Microprocessor Architectures: a Comparison Based on Code Generation by Compiler." Communications of the ACM 29, no. 10 (Oct.): 978-990.

more than 4 instructions, indicating a predicted performance of the prototype of 4 MIPS on average (assuming 30 cycles per interpreted instruction). This is three times faster than a purely interpreted approach.

Future possibilities

The following list contains some possible future extensions to improve performance:

- Unrolling. A small inner loop can be unrolled in the translated code, just as an optimising compiler would do.
- Peep-hole optimisation. This is particularly interesting if the instruction sets differ considerably.
- Event timer arithmetic. In the case of a simple loop that takes N cycles, then the entry code can determine how many iterations of the block to perform. This is faster than decrementing with N each time on some architectures (especially if there is a decrement-and-branch-if-positive instruction).
- Memory translation optimisation. If the entry code can assert that some of the memory translations will always be from the same page within the block, then only a single entry need be checked. Indeed, code in the block can be hard-coded to directly access host memory if this can be done while maintaining a consistent view of memory.

Conclusion

We have discussed various approaches to instruction-level simulation of a target architecture. The conclusion is that an interpretive approach has a high level of functionality, which we would like to keep, and that direct execution is the fastest approach. We proposed *partial translation* as a hybrid of the two, where we can translate only simple sequences of code, and handle any complicated situations in interpreter routines.

We presented a prototype that shows that the design can be effectively implemented. Generation of code is sufficiently cheap to do in run-time. Some early results indicate a substantial performance improvement over a purely interpretive approach, with no loss of functionality.

Bibliography

Barbacci, M. R. 1981. "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications." IEEE Transactions on Computers C-30, no. 1 (Jan.): 24-40.

Bedichek, R. 1990. "Some Efficient Architecture Simulation Techniques." In USENIX - Winter '90, p53-63.

Bell, J. R. 1973. "Threaded Code." Communications of the ACM 16, no. 6 (June): 370-372.

Calingaert, P. 1979. Assemblers, Compilers, and Program Translation. Computer Science Press, Rockville, USA.

Canon, M. D. et al. 1980. "A Virtual Machine Emulator for Performance Evaluation." Communications of the ACM 23, no. 2 (Feb.): 71-80.

Deware, R. B. K. 1975. "Indirect Threaded Code." Communications of the ACM 18, no. 6 (June): 330-331.

Ditzel, D. R.; A. D. Berenbaum. 1987. "Using CAD Tools in the Design of CRISP." IEEE Design and Test, June 1987.

Fujimoto, R. M. and W. B. Campbell. 1988. "Efficient Instruction Level Simulation of Computers." Transactions of the Society for Computer Simulation 5, no. 2 (Apr.): 109-124.

Glass, R. L. 1980. "Real-Time: The 'Lost World' of Software Debugging and Testing." Communications of the ACM 23, no. 5 (May): 264-271.

Hagersten, E. 1992. "Towards a Scalable Cache Only Memory Architecture." PhD thesis, Royal Institute of Technology, Sweden (SICS Dissertation Series 08, Swedish Institute of Computer Science, Sweden).

Hagersten, E.; A. Landin; and S. Haridi. 1992. "DDM - A Cache Only Memory Architecture." IEEE Computer 25, no. 9 (Sept.): 44-54.

Huguet, M.; T. Lang; and Y. Tamir. 1987. "A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements." In Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques (St. Paul, Minnesota, June 25-26). 14-25.

```
(e)     0x3a810  <end+93512>:    sethi   %hi(0x12ef000), %i0
        0x3a814  <end+93516>:    srl   %i0, 0xa, %i0
        0x3a818  <end+93520>:    add   %i0, %l6, %i0
        0x3a81c  <end+93524>:    andn   %i0, 0xffc, %i1
        0x3a820  <end+93528>:    subcc   %i1, %l5, %g0
        0x3a824  <end+93532>:    bne   0x3a878   <end+93616>
        0x3a828  <end+93536>:    sethi   %hi(0x25000), %i2
        0x3a82c  <end+93540>:    st   %i1, [ %i0 + %i2 ]
(g)     0x3a830  <end+93544>:    sethi   %hi(0), %l4
        0x3a834  <end+93548>:    sethi   %hi(0x10000), %l6
        0x3a838  <end+93552>:    add   %l4, %l6, %l3
        0x3a83c  <end+93556>:    add   %l3, %l6, %l2
        0x3a840  <end+93560>:    add   %l2, %l6, %l1
(h)     0x3a844  <end+93564>:    sethi   %hi(0x41000), %i0
        0x3a848  <end+93568>:    ld   [ %i0 + 0x90 ], %g1
        0x3a84c  <end+93572>:    ld   [ %i0 + 0x94 ], %g3
        0x3a850  <end+93576>:    add   0x48, %g4, %g4
        0x3a854  <end+93580>:    add   8, %g4, %g5
(i)     0x3a858  <end+93584>:    st   %l6, [ %g7 + 0x38 ]
        0x3a85c  <end+93588>:    st   %l4, [ %g7 + 0x68 ]
        0x3a860  <end+93592>:    st   %l3, [ %g7 + 0x6c ]
        0x3a864  <end+93596>:    st   %l2, [ %g7 + 0x70 ]
        0x3a868  <end+93600>:    st   %l1, [ %g7 + 0x74 ]
        0x3a86c  <end+93604>:    st   %l7, [ %g7 + 0x7c ]
(j)     0x3a870  <end+93608>:    jmp   %g1
        0x3a874  <end+93612>:    nop
(f)     0x3a878  <end+93616>:    st   %l6, [ %g7 + 0x38 ]
        0x3a87c  <end+93620>:    st   %l7, [ %g7 + 0x7c ]
        0x3a880  <end+93624>:    sethi   %hi(0x41000), %i0
        0x3a884  <end+93628>:    ld   [ %i0 + 0xe0 ], %g1
        0x3a888  <end+93632>:    ld   [ %i0 + 0xe4 ], %g3
        0x3a88c  <end+93636>:    add   32, %g4, %g4
        0x3a890  <end+93640>:    add   40, %g4, %g5
        0x3a894  <end+93644>:    jmp   %g1
        0x3a898  <end+93648>:    nop
```

The code is an actual disassembly from within GDB, slightly modified for clarity. The layout of the code is sketched in figure 5.

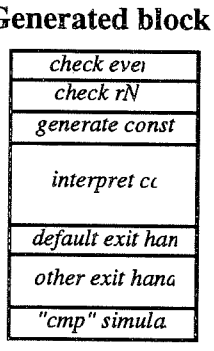

Figure 5 - Layout of generated code

Some comments to the code.

By now we know that the code will be executed at least once. The first thing to do is therefore to read the registers that are accessed in the block (as source registers). Note that we do not optimise away registers that are overwritten in the block before they are used.

At "lbl_1", we now begin to simulate the first instruction, which is a read. The memory accesses are optimistic—the code will only handle the simplest case. The memory simulation is presumed to cache MRU (most recently used) read/write TLB entries in two pairs of registers: (rLP_R, rPP_R) and (rLP_W, rPP_W). Better performance might be gained by also encoding a first-level hash table lookup.

Assuming that there is no problem, we do the read. If there is a problem, we perform exit 1. This updates the rSTATUS register to indicate that the translated block is performing poorly. Alternatively, we could update a block-specific status field. The block could then deallocate itself and, upon re-generation, the translation routine has more information on how the block will behave and can generate different code.

By combining information on what code is in the translated block, and as well picking up information as described above during runtime, the translation routine has information available for very sophisticated code generation schemes. For instance, it might detect a copying or zeroing block (very common operation), and generate code to detect virtual memory translation misses only once for each memory page involved in the copy/zero operation.

Next we carry out the write in a similar manner to the read. Error handling is the same as for reads.

The next two instructions in the block can be translated to two host instructions: the add instructions cannot generate exceptions and do not access memory.

Next, the conditional branch to the beginning of the loop is simulated by two host instructions. If we decide to try the loop again, we branch to "lbl_2". Here, we check for events, and, if none occur, repeat the loop. If an event will occur, we write back the registers, update the status register, and (by jumping to exit_0), interpret the instruction instead.

If the loop has finished, we set status bits, update event counter, update program counter, and jump to *exit,* which does a normal clean-up: writes back registers, checks for event at the end of execution, and dispatches the next instruction.

In all cases, the code will exit with rEVENT, rXIP, rNIP, rOP and all CPU registers in a correct state.

The code above has 17 instructions in its central loop and no more data reads and writes than the original code has (one of each). Entry and exit code is 23 instructions. This yields a potential slow-down close to 3 (recall that the original code had 6 instructions including one read and one write).

Handling status flags

A classic problem with code generation is how to handle status flags. In the case of the M88100, the corresponding problem is the "cmp" instruction. This compares two registers and stores a vector in a destination register, based on the relation between the two. Typically, this destination register is only used once in a subsequent "bb" (branch on bit) instruction. If register containing the bits is subsequently overwritten, regardless of outcome of the branch, then we need only simulate the branch-on-condition function.

Our approach is to assume that a cmp/bb pair can handled thus, and to note that we've done so. Later we can determine whether our guess was correct. Compensating a bad choice can still be done in single-pass code generation by branching in and out from the point in the code to new code that does a correct "cmp".

Selection of what to translate

A problem that remains is how to select the portions that should be compiled. We have two extremes: completely interpreted code, and completely compiled. Conceivably we have a contiguous scale in-between.

We would want to compile as much as possible: essentially all critical loops. Compiling is not necessarily expensive. A straight-forward approach is to translate each MC88100 instruction to one or more SPARC instructions. An advantage of avoiding a more sophisticated approach is that one-to-many translation is fast and relatively simple to implement.

Anything that would require relatively complex processing - such as memory accesses that miss the TLB, or updating the event queue - would require so much saving of state that we might as well exit the compiled code. This would include most events.

The first case may involve jumping to the service routing in the interpreter for that instruction (if it is complex) or completing that instruction (if it is simple). By updating all simulator status necessary, no other state needs to be considered on exit.

As with interpretive decoding, the generated code must be sufficiently general to be used on all future entries to the code. For instance, it should not depend on target processor, in the case of a multiprocessor.

Maintaining a fine granularity of event handling is an important function of the simulator. This requires that compiled code blocks check for events. One approach is to maintain a coarse precision inside the compiled code and to execute interpreted code when a compiled block risks treating an event imperfectly. This means that the entry code in a compiled block needs to determine whether it is "safe" to continue. If it isn't safe, the block executes only the first instruction and then dispatches the next instruction.

Partial translation—an example

The following (hand-coded) example illustrates what a target code block might look like. The block assumes no entry/exit being done by a TRANSLATE instruction, so either the block is dispatched directly or a TRANSLATE instruction only serves as indirection (and perhaps some sanity checks). The code being translated is the example in the previous section.

After the example code, we will describe in detail how it works.

```
block:  r1 = cmp rEVENT,8          ; can we do at least one iteration?
        br  1,r1,<exit_0>          ; no
        cmp rSTATE,<CONST>         ; legitimate CPU state?
        br  ne,<exit_d>            ;  no, abort and de-allocate
        cmp rNIP,<CONST>           ; are we in a branch delay slot?
        br  ne,<exit_0>            ;  yes, dispatch to service routine
        r1 = rCPU[20]              ; load the registers used ... r5
        r2 = rCPU[27]              ; ... r7
        r3 = rCPU[36]              ; ... r9
        r4 = rCPU[52]              ; ... r13
lbl_1:  r5 = shift r1              ; check vs cached transl. for read
        cmp r5, rLP_R
        br  ne,<exit_1>            ; we can't handle read
        r5 = mask r1               ; calculate physical location (in sim)
        r2 = rPP_R[r5]             ; do the read
        r6 = shift r3              ; check vs cached transl. for write
        cmp r6, rLP_W
        br  ne,<exit_2>            ; we can't handle write
        r6 = mask r1               ;  physical location (in simulator)
        rPP_W[r6] = r2             ; do the write
        r1 = r1 + 4                ; increment both, nothing can happen
        r3 = r3 + 4
        cmp  r4,r3                 ; should we loop again?
        br  ne,<lbl_2>             ;  yes, try to do it again
        rSTATUS = <CONST>          ;  no, set status bits correctly
        rEVENT = rEVENT - 8        ; loop took 8 cycles (we assume)
        rPC = rPC + 24             ; point to next instruction
        br <exit> ; clean up

                                   ; try to do loop again
lbl_2:  rEVENT = rEVENT - 8        ; update event counter
        cmp rEVENT,8               ; can we do one more iteration?
        br  ge,<lbl_1>             ; yes, and registers already read in
        rCPU[20] = r1              ;  no, restore registers
        rCPU[28] = r2
        rCPU[36] = r3
        rCPU[52] = r4
```

Partial translation

We wish to combine the performance benefits of direct execution (running generated native code) with the flexibility and accuracy of interpretation. Our experiences with interpretive simulators indicated that the most of the work carried out was very simple. It is then natural to consider how to handle the common cases in generated code, while maintaining the semantics of a instruction-level interpreter. These semantics include:

- Accurate handling of asynchronous events. An event specified to occur on a particular cycle must occur exactly after the instruction executed on that cycle. This permits correct statistical sampling of application behaviour, correct simulation of asynchronous devices, and exact user breakpoints for debugging.
- Correct processor state. The target processor volatile state (registers and status flags) must be correct *whenever so required.*
- Correct memory access sequence and timing.

Incremental translation translates object code to an internal format that is more easily interpreted. Partial translation extends this by allowing one or more target instructions to be translated to host instructions, while maintaining the functionality and correctness of the original design. The choice of which and how many instructions to translate can be done according to several heuristics. For the purposes of discussion, we assume that these heuristics will, statically or dynamically, detect a simple inner loop.

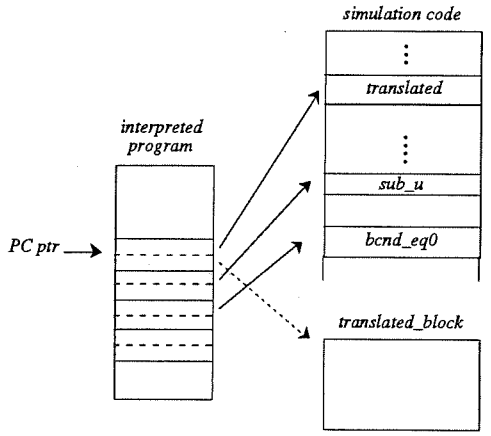

Figure 4 - Partial translation

Figure 4 illustrates how of interpreting an intermediate format can be combined with direct execution of generated code. Assume that we wish to translate a sequence M88100 instructions to native SPARC code. The intermediate code is in a 64-bit format, where the first 32 bits points to the code that simulates the corresponding instruction. The second 32 bits contain parameters for the instruction. Each 88k instruction is translated to this format.

When execution of the program reaches the first instruction in the block, the service routine being jumped to is actually a run-time generated block of SPARC code. When this block has completed, it will dispatch the first instruction after the block. "Dispatch" here means reading the intermediate format and jumping to the pointer containing the first 32 bits.

Consider the following example code:

```
label:    r7 = [r5]          ; read next word from source
          [r9] = r7          ; store it to destination
          r5 = r5 + 4        ; increment source pointer
          r9 = r9 + 4        ; increment destination pointer
          cmp  r13,r9        ; finished?
          br ne,<label>      ; if not, continue
```

instruction; instead, the target volatile state can be fully or partially mapped onto the host. The performance potential is promising: as fast as half real time has been demonstrated.

There are several approaches to block translation. The major distinction is the form of code available: HLL source code, assembly code, or object code.

If the source code is available, a similar compiler can be used to generate target and host code. If the basic blocks in the two resulting files have a one-to-one relationship, any statistics that are on the block level can easily be extracted from the target code and inserted into the host code: the result is very fast native code.

If the assembler code is available, assembler-to-assembler translation does not require any flow-control analysis. The assembler can either be translated to C and re-compiled, or it can be directly translated (Fujimoto et al 1988; Mills et al 1991; Huguet et al 1987).

Finally, if only object code is available, translation must be put off until run-time: there is no *a priori* manner of distinguishing text from data (May 1987).

Interpretive translation

The third approach of second-generation simulators is interpretive translation: target code is translated into a format that is easier to interpret, and this format is then interpreted. By using a simple format, the interpreter can work efficiently.

Interpretive translation gets a performance boost by threading the interpreted code. Threaded code was first described by Bell as a general programming method (Bell 1973). Bell distinguished between hard code, interpretive code, and threaded code. Figure 3 illustrates the three code types. Hard code is the traditional type, i.e., program flow runs along a main segment, occasionally doing a subroutine call.

Interpretive code has a central control segment, the interpreter, which calls various routines on the basis of the interpreted code. In threaded code, the task of the interpreter is done by the service routines. Instead of returning to a central interpreter, the epilogue of each service routine fetches sufficient information from the threaded code to be able to immediately jump to the corresponding service routine. There are several variations of threaded code, including indirect threaded code (Deware 1975).

Interpretive translation has many strengths. To begin with, there is a continuous representation of the processor state. This means that exceptions, virtual memory, and other system software mechanisms are readily simulated. Furthermore, the small semantic distance between the translated format and the target code means that the overhead of translation virtually disappears.

Furthermore, interpretive translation can be extended to efficiently support multiprocessors and can be written to closely match the memory bandwidth requirement of target machine programs (Magnusson 1993).

Drawbacks of generating code

The principal advantage of block translation is performance. Instruction set interpreters have a large overhead due to decoding, dispatching, and saving/restoring the volatile state. Within blocks, the volatile state can be fully or partially mapped to the host processor, and no decoding or dispatching needs to be done within the block.

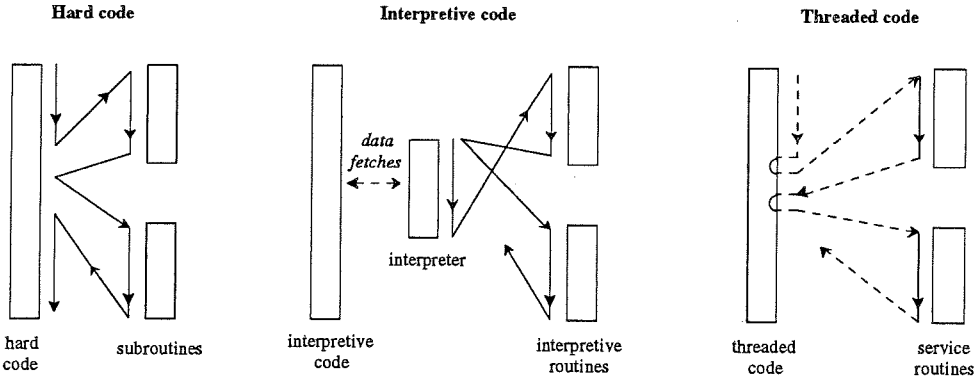

Figure 3 - Programming methods.

important tool in system software development, both in studying operating system behaviour and in developing system software for next-generation hardware, in application tuning, and in execution of foreign code.

Simulating a multiprocessor presents some special problems, notably code expansion and efficient time slicing of processors. Also, modern processors have aggravated the memory bottleneck, and the internal formats used by a simulator must be compact.

The organisation of this paper is as follows. First, we discuss in more detail the terminology and previous work in the field. Next we describe partial translation. Then we present an extensive example of translated code that a basic translation routine might generate. Finally, we present a prototype implementation of the technique and some early experiences.

Terminology and Definitions

The distinction between developing and analysing is important. Essentially, developing hardware means mainly getting it to work. This implies that the simulator needs to be accurate but not fast: long-term problems can be handled on the real hardware. Accuracy in simulating hardware is generally both difficult and expensive, and several approaches exist - however, accurate hardware simulation is beyond the scope of this paper.

Analysing hardware or system software, or developing system software, is quite different from developing hardware: efficiency in computer resource requirements is necessary to allow for realistic benchmarks or test suites, or to keep a reasonable edit-compile-execute cycle. This means that, of necessity, the hardware needs to be abstracted to a more functional level.

We will distinguish between *behavioural* simulation and *performance* simulation (Fujimoto *et al.* 1988). The former refers to simulating the hardware sufficiently well for programs to run with the same results, the latter states that the execution time on the target machine should also be accurately simulated. Performance simulation implies that aspects of the hardware such as TLBs, caches, pipelines, and memory latencies all need to be accounted for.

The distinction between behavioural and performance simulation is not rigid, however, since a program's behaviour may be affected by timing.

We're only interested in the typical instruction-set simulation model. Processor state that is visible to a program (e.g. general registers, condition codes, and memory) is represented by variables and the fetch-decode-execute cycle of the processor is implemented with a program loop.

We distinguish between *user-level* and *system-level* simulation. User-level simulation means that programs that run in the target's equivalence to user mode will run correctly. This means that the simulator needs to provide the operating system services, including any system calls used by the program.

System-level simulation, in contrast, will execute user-level programs but running within the context of the system, i.e., the system software must also be running on the simulator. In both cases, when a complete hardware system is simulated we speak of a *virtual machine* (Canon *et al.* 1980). Figure 1 shows the matrix of major simulator categories.

User-level simulation is considerably easier to implement than system-level simulation, since aspects of the target hardware like I/O, virtual memory, and multiprogramming can be ignored. Similarly, behavioural simulation is easier to implement than performance simulation. In this case, aspects like cache, pipelines, and memory latencies can be ignored.

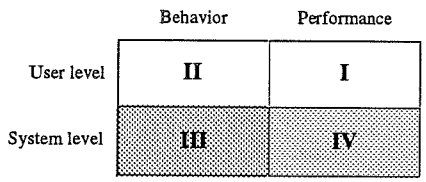

Figure 1 - Simulator matrix